Confidentiality and Integrity in Distributed Data Exchange

Gerome Miklau

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2005

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Gerome Miklau

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

_____

Dan Suciu

Reading Committee:

_____

Alon Halevy

_____

Dan Suciu

_____

John Zahorjan

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Confidentiality and Integrity in Distributed Data Exchange

Gerome Miklau

Chair of the Supervisory Committee:
Associate Professor Dan Suciu
Computer Science and Engineering

The distributed exchange of structured data has emerged on the World Wide Web because it promises efficiency, easy collaboration, and—through the integration of diverse data sources—the discovery of new trends and insights. Along with these benefits, however, there is also the danger that exchanged data will be disclosed inappropriately or modified by unauthorized parties. This dissertation provides conceptual and practical tools for ensuring the confidentiality and integrity of data that is exchanged across heterogeneous systems.

Securing data in such settings is challenging because participants may behave maliciously, and because their remote systems are outside the control of the data owner. This dissertation addresses these challenges, first by developing a precise analysis of the information disclosure that may result from publishing relational data. This is a critical prerequisite to forming a policy for permitting or denying access to data. The novel notion of information disclosure presented here can capture leaks that may result from collusion by multiple parties, or from prior knowledge they may possess. This dissertation then addresses the practical problems of safely and efficiently guaranteeing security properties for distributed data. To provide confidentiality, a flexible fine-grained encryption framework is proposed which allows data owners to construct, from a set of access policies, a single encrypted database that can be stored and exchanged by all parties. Access is granted by separately disseminating keys. To provide integrity, an efficient authentication mechanism is described

which can be used to detect tampering when data is stored by an untrusted database. Together these techniques can significantly advance the security of distributed data exchange.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Number Page

ACKNOWLEDGMENTS

This work would not have been possible without the guidance and insight of Dan Suciu. He is a gifted teacher, a patient and generous advisor, and a brilliant researcher. He has taught me precision and rigor, and he has shaped my graduate experience by working closely with me and sharing the process of research. I hope to bring these gifts to my teaching and research in the future.

I am also grateful to Alon Halevy for first sparking my interest in databases, as well as for the ongoing support of my research. I thank Phil Bernstein for his excellent advice and thorough feedback throughout the process, and I am grateful to John Zahorjan for his participation on my committee.

My work has benefitted greatly from the support of my colleagues in the UW database group Anhai Doan, Zack Ives, Rachel Pottinger, Igor Tatarinov, Peter Mork, Jayant Madhavan, Ashish Gupta, Nilesh Dalvi, and Xin Luna Dong, as well as my friends Don Patterson and Luke McDowell. Many of the positive professional relationships mentioned above are a consequence of the unique supportive environment of the University of Washington Computer Science department. The faculty and staff of the department, each of whom contribute to that environment, therefore also deserve my thanks.

My parents have provided unending support and encouragement, for which I will always be grateful. In addition, my grandfather, Fred F. Tirella, valued higher education and instilled that value in his children and grandchildren. He has made a great contribution to this work by supporting my education financially, in his life, and after his death in 1999. I am also grateful to my aunt and uncle, Eileen and Edward Kostiner for their support.

Chapter 1

INTRODUCTION

The technological advances of the past two decades have resulted in an unprecedented *availability* of data. This is largely the consequence of continuous improvements in computer hardware, the rapid evolution data management technologies, and the attendant explosion in data collection. Hardware advances have provided massive persistent storage, extremely fast networks, and powerful processors. Fundamental advances in data management have provided systems to manage massive data sets and millions of transactions. And importantly, technologies for the integration of data sources and distributed data exchange have allowed the combination of information sources and the collaboration across geographic and organizational boundaries.

These increased technical capabilities have in turn enabled an explosion of data collection. There are now stored electronic records of many events, actions, and relationships in the real world. For example, data about individuals is collected through traces of financial transactions, phone calls, web browsing histories, library loans and video rentals. Environmental monitoring systems collect and store data about the physical world and, in some cases, the location of individuals.

The unprecedented availability of data undoubtedly offers a great *promise*. Information we need is often easy to find, and conveniently accessible. We can combine and integrate data to derive new insights and discover new trends. We can share and exchange information to enable collaboration between distant parties.

But along with the promise, these capabilities also present a possible *peril*. The peril results from the fact that the misuse of data can cause harm, and that despite recent data management advances, there is the danger that we will lose control of information about

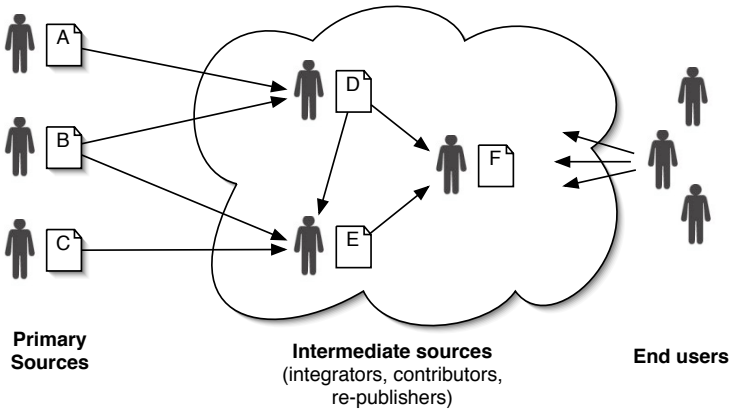

Figure 1.1: Distributed data exchange

us, or information owned by us. For example, in the first half of 2005 alone, nearly 50 million records containing sensitive data about individuals have been lost or stolen from corporations, government agencies, or academic institutions [31]. A more subtle example of the possible peril of current data management practices was demonstrated by a privacy researcher who integrated two public databases, each deemed safe for release in isolation, to reveal the medical diagnosis of a prominent politician [127].

The guiding objective of this dissertation is to provide conceptual and practical tools to enable the safe exchange and sharing data—that is, to avoid the peril and achieve the promise.

1.1 Problem setting

In many modern applications, the classical client-server processing architecture has been supplanted by more distributed architectures, characterized by many parties interacting across heterogeneous systems. The distributed data exchange scenario that is the setting for this thesis is illustrated in Figure 1.1. It may include many primary sources, many intermediate publishers, and many end-users. The primary sources publish data, the intermediate participants combine, transform, and modify the data, and the end-users consume the data. Examples of applications where data is published and exchanged in this man-

ner include scientific data management, e-business, and medical information management, among many others.

The data exchanged in such settings is often sensitive—that is, its misuse could cause harm. For example, social security numbers are considered sensitive data in the United States because they can be used by a malicious party to open a false bank or credit card account. The results of a medical exam performed on a patient constitute sensitive data—not only because the information should remain secret, but also because the exam results may be used to make critical medical decisions. Modification of the exam results could therefore have harmful consequences by impacting those decisions. The types of misuse considered here fall into two broad categories: unauthorized disclosure and unauthorized modification. These correspond to two classical security properties of confidentiality and integrity.

A data owner has two competing goals (roughly corresponding to the promise and peril mentioned above). First, the data owner wants to permit legitimate use of the data by a potentially large group of authorized users, for example to satisfy business need and enable collaboration. Second, the data must be protected from unauthorized disclosure and unauthorized modification. Recipients of data are interested in efficient access to data they are permitted to see, and are also concerned with the authenticity of that data.

The fundamental challenge of providing safe and secure data management in such scenarios stems from the fact that they are built upon heterogeneous systems, owned and administered by different parties. These systems may consist of conventional relational database management systems, web servers, and middleware systems. Although we assume a data source has trusted systems used to store and process the data, they have no control over how the data will be processed once it is published.

This thesis focuses on two of the classical problems of security research: secrecy of data and integrity of data. However, there are a number of distinguishing challenges faced here. First, the data is managed in a fine-grained manner. The types of data managed include collections, relations and structured documents, whose constituent parts must be managed safely. In other settings, security research considers uniform blocks of data and some problems are simplified. Secondly, as mentioned above, when data is exchanged, it

leaves the centralized, trusted domain of the data owner, and will be processed by client systems.

1.2 Overview of the solution space

The ultimate solution to the data security challenges outlined above is likely to require a wide range of technologies and societal conventions, including contributions from the following domains:

- *Physical security* The first line of defense is physical security: protecting data storage devices from theft, loss, or destruction by using locks, alarm systems, etc. Approximately 60% of the personal data mentioned above was lost because of breakdowns in physical security.

- *Secure systems, networks, and protocols* Security engineering of systems and networks aims to protect resources in a multi-user environment, including identifying authorized parties, negotiating access to resources, and rights to perform operations. The security of a database management system depends on the security of the underlying operating system, and also presents its own challenges and vulnerabilities.

- *Cryptography* Cryptography has been informally described as a means for "taking trust from where it exists to where it's needed" [92], and it plays an important role in meeting our challenges, since we cannot trust recipients' systems. Cryptographic techniques are used here to provide secrecy of data and to detect tampering when data is released outside the data owner's realm of control.

- *Modeling the adversary* Securing data depends on accurately modeling the capabilities of adversaries, which includes understanding the resources (money, time, computation) an adversary may have. Further, the adversary's knowledge is one of the most difficult "resources" to model, as is the adversary's ability to infer new facts from known facts.

- *Regulation* Legislation and government regulation has an enormous impact on how data is stored, processed, and exchanged, and can mitigate the risks of security breaches. In some cases the need for enforcement by technical means may be avoided. For example, the Health Insurance Portability and Accountability Act (HIPAA) [77] controls by legal means how data can be collected and used, and under what conditions it can be released to medical professionals. Such policy enforcement across the medical community would be virtually impossible using security technology alone.

Successfully securing distributed data depends on the careful integration of each of the factors above. Physical security and regulation are beyond the scope of this thesis. The techniques explored here include system and protocol design, cryptography, and also database theory and logic for understanding disclosure and inference by the adversary. A thorough background discussion on these and related topics is provided in Chapter 2.

1.3 Overview of contributions

Managing disclosure

Chapter 3 addresses the problem of *determining what data can safely be published*. The answer to this question is a critical prerequisite to forming a policy for permitting or denying access, and requires a precise understanding of the information disclosure that may result from publishing data. Analyzing disclosure is difficult because confidential facts may be unexpectedly inferred from published data, and also because authorized recipients may collude to make unauthorized discoveries.

Our analysis of disclosure is applied to data stored in relational form, prior to publishing. In such systems, the basic object of protection is the relational view. To formulate the disclosure problem, we assume the data owner has a sensitive query, the answer to which must remain secret. We propose and analyze a new theoretical standard for determining when it is safe to publish the view without disclosing information about a sensitive query.

As an example, a simple view over medical data might include patient name, age, and gender for those patients treated by a particular physician (omitting other attributes such

as disease and blood-type). This view can be used to derive some query answers about the original database, but not others. The use of a view for answering queries has been studied extensively in the database community. However, the extent to which the view may leak information about a sensitive query has not been well understood. In particular, although the user (or an adversary) may not be able to compute the exact query answer from the view, the answer may be inferred with high likelihood, or other partial information may be revealed.

Chapter 3 describes a novel standard of *query-view security*. When a view and a privileged query are secure, the view contains no information about the answer to the query, and the user will have no advantage in computing or guessing the answer. Intuitively, a query and view are deemed secure if the user's *a priori* knowledge about the query is the same as the user's knowledge about the query having seen the view and its answer. To measure and compare knowledge, we assume a probability distribution over possible databases and query answers.

For simple relational queries and views, we show that it is possible to decide query-view security, and provide tight bounds for the decision problem. The definition of query-view security has some surprising consequences. For example, it shows that simply removing sensitive columns from a relational table—a common strategy in practice—does not totally protect the sensitive data. These results can also account for some forms of knowledge the user may already have, resulting in a notion of relative query-view security. This work originally appeared in the proceedings of the Conference on Management of Data, 2004 [104] and has been accepted for publication in the Journal of Computer and System Sciences.

Confidentiality in data exchange

Chapter 4 addresses the problem of *how to efficiently distribute data while adhering to access control policies*. Having determined which views are safe for publication, there may be many users whose authorized views are highly overlapping. For example, the view of patient data above may be defined for each physician in the hospital. It may be inefficient to transmit separate views to each user. Additionally, once published, the data owner typically

relinquishes all control over the user's further processing of data.

Chapter 4 describes a framework in which the data owner begins with an access control policy, specifying which parties shall have access to which data. A single partially-encrypted version of the database—to be used by all users—is automatically generated to enforce the access control policy. The encrypted output is represented as a structured XML document. As a simple example, in the case of hospital data, a single database instance could be constructed that includes all patient data. It could be published and disseminated freely, but would be encrypted so that physicians could only access data for patients under their care. In this framework, control of the data is accomplished by transmitting sets of keys to users. Key transmission can be done all at once, or in more complex interactive protocols as users meet requirements (e.g. payment for access), or satisfy threshold constraints (e.g. fewer than 5 items accessed).

Using these techniques, data owners can efficiently publish their data with more control over its use. The framework is also important because it is an automated way to generate encrypted data with precise access semantics, while not requiring the user to directly invoke cryptographic routines. Before encryption, the access control policy is resolved into a logical model of data protection, where keys "guard" access to parts of the structured document. The logical model has precise semantics, and can be optimized to improve the efficiency of the output while respecting the declared access policy. This work appeared in the proceedings of the Conference on Very Large Databases, 2003 [102].

Integrity in data exchange

In Chapter 5 we turn from confidentiality concerns to integrity, and address the problem of *how recipients of data can be assured of its authenticity.* We present a vision for managing the integrity of data in distributed settings using cryptographic annotations that offer strong evidence of origin authenticity.

Hash trees are a key technology for supporting such annotations, and can also be used to provide integrity of data in conventional database systems that may be vulnerable or untrusted. Although hash tree techniques are well-known, they do not always permit ef-

ficient operations, particularly in a database system. The emphasis of this chapter is on implementation techniques which demonstrate the feasibility of hash trees in databases. We show that relations can be used to store the hash tree so that verified queries and updates can be performed efficiently, while preserving the other benefits of a relational database. Using novel strategies for data representation and indexing, we show that data integrity can be verified with modest computational overhead in a client-server setting. In addition, the architecture of the system makes this a versatile solution—it can be easily adapted to provide integrity guarantees for any relational database system.

Together, the techniques developed here can significantly advance the security of data shared and exchanged in modern distributed settings. The sequel is organized as follows. In the next chapter a thorough background of related topics is presented including access control, cryptographic tools, and disclosure analysis. The major contributions described above are included in the subsequent three chapters, followed by concluding remarks in Chapter 6.

A note on data models

A data model is the high-level, organizing representation for information stored in a database. This dissertation focuses on two data models: the classical relational data model, as well as a semi-structured data model where data is represented in XML form. Addressing both data models is critical for distributed data exchange. Prior to publishing, data is likely to reside in relational systems since these are the best-performing systems for large scale data management. But when relational data is published it is often represented as XML [120]. Intermediate storage in data exchange settings may use either relational or native XML systems. The native storage of XML is an active research area [42, 66, 81], with several commercial products availabile [137, 128, 8]. Nevertheless, relational systems are much more common, and XML data is often decomposed and stored in relational systems [65, 121]. (See [117] for a recent survey of how XML data is managed in relational systems.)

Each of the problems addressed in this dissertation is posed in terms of the most rele-

vant data model. The analysis of disclosure is applied to relational data. The protection mechanisms using cryptography focus on XML data. Although managing integrity for data exchange is most appropriate for XML data, we addresss in Chapter 5 a particular implementation problem assuming that intermediate storage is relational.

Chapter 2

BACKGROUND

This chapter is a background discussion of three areas of security research that underlie the contributions of this thesis: access control, cryptographic techniques, and disclosure analysis. Access control is the starting point for security in computer systems. Below we review general concepts of access control, and describe the standard capabilities of access control in databases. In distributed data exchange, access control has two basic limitations. First, it requires enforcement by a trusted authority. Second, while it protects direct access to data, it does not necessarily prevent partial disclosures and inferences. Both limitations are investigated in this chapter, with an emphasis on how cryptographic techniques can be used to address the problem of trusted enforcement, and a discussion of the importance of disclosure analysis to understanding the real consequences of data publishing. We preface the background discussion with a brief overview of security properties.

2.1 Security properties and threats

The goal of this dissertation is to provide the two classical security properties of confidentiality and integrity. We define these notions below, and relate them to the more complex notion of personal privacy.

Confidentiality is a guarantee that data has not been disclosed to an unauthorized party. Threats to confidentiality include the direct release of sensitive data values, approximate disclosures, and leaks resulting from inferences and outside knowledge. One way to provide confidentiality is to enforce access controls, which permit or deny access to data items. Another way to provide confidentiality is through encryption, often used to protect data on insecure networks or storage devices.

Integrity is a guarantee that data has not been modified from its original state by an unauthorized party. This requires both that data come from an original source, and that

it be unaltered from its original state. Data authenticity is a closely related term, but does not always include *freshness*: a guarantee that a message or data item is current, and is not being re-used outside its original context. Throughout the discussion we use integrity as the most general term (including both data authenticity and freshness) and we distinguish between these particular properties where necessary. Threats to data integrity include tampering, message replay, and accidental data loss. Tampering can include modification or deletion of existing data, or insertion of new data items. Once again, data integrity can be provided though access control, and also by cryptographic means. Access control protects data integrity by limiting who can alter data. Digital signatures and cryptographic hash functions can protect against the threat of tampering (but not data deletion).

Data privacy is a term used synonymously with data confidentiality. Personal privacy, however, is a more complex notion with no single definition. Legal conceptions of privacy have been grounded in the right to be left alone [133], and the freedom from being brought to the attention of others [70]. Most relevant to our discussion is the concept of *informational privacy* [38], described by Westin as the ability to "determine for ourselves when, how, and to what extent information about us is communicated to others" [134]. Clearly mechanisms for data confidentiality are required to protect an individual's informational privacy. Integrity of data is also critical to informational privacy, since there is little use in controlling the information communicated to others if that information can be tampered with. While this dissertation is focused strictly on data confidentiality and integrity, our contributions support informational privacy when they are applied to personal data.

2.2 Access controls

Access controls are a basic tool of computer security. Because this dissertation addresses two fundamental limitations of access control for data exchange, we review here basic notions of access control relevant to all computer systems, and then focus on database access control, for both relational and XML data.

2.2.1 Access control in computer systems

Access controls regulate direct access to resources in a computer system. An access control policy consists of rules referring to *subjects*, *objects*, and the *privileges* granted to subjects on objects. Subjects are the active parties, usually users identified by name, or by role. Objects are system resources. In general, they may include network access, processor time, or memory, but the objects of concern here are data items of varying granularity. In file systems, data objects are files and directories. In relational databases, objects may be tables, columns, attributes, or defined views. For semi-structured data in XML form, objects may be elements, attributes, or subtrees. Privileges over data objects typically include operations like *create*, *read*, *modify* or *delete*. Ensuring that only authorized subjects have read access to objects provides confidentiality. Ensuring that only authorized subjects have create, modify and delete privileges is central to data integrity.

An access control policy can be represented using a matrix which stores for each subject and object a set of privileges. A column in the matrix (typically called an access control list) refers to one object and lists each subject and their privileges. Limited access control lists are stored with files in the Unix operating system. A row in the access matrix refers to one subject, and lists all objects and privileges held by that subject. Such a list is typically called a capability list [40, 136]. In some systems a capability is an electronic token naming an object and a set of privileges. Possession of the capability authorizes a subject to perform the indicated operations on the named object. (The access control mechanism for distributed data, described in Chapter 4, is conceptually similar to a capability-based system since the cryptographic keys granting access to select portions of an XML tree act like capabilities.)

An access control policy must be carefully enforced. Subjects must be authenticated[1] and their requests for access to objects permitted or denied by a monitoring process. This access monitor is a *trusted* system component: its failure can break the security policy. Therefore, the code for the access monitor must be tested and verified. In addition, privileged users (e.g. system administrators or database administrators) responsible for configuring access policies, act as trusted parties in the system.

---

[1] An entity is authenticated if, by providing evidence, the system is convinced of its true identity. [94]

2.2.2 Access control in database systems

We review below the conventional capabilities of access control in databases for relational data, XML data, and statistical databases. In the relational model, the objects protected may be tables, columns of tables, or derived tables that are defined using logical conditions (i.e., views). For XML data, protection is often negotiated at even finer granularity, at the level of individual data items, or elements. Modern relational systems implement their own access control mechanisms. File system access controls provided by the operating system are generally insufficient because one or more tables may be stored as a large file, and database access control must be applied at a finer granularity.

Relational database systems implement access control in the SQL language, using the `GRANT` and `REVOKE` commands, which were heavily influenced by the authorization mechanism in System R [74]. The `GRANT` command is used to give privileges to users. It has the following syntax:

`GRANT` *privileges* `ON` *object* `TO` *users* `[WITH GRANT OPTION]`

In SQL, *object* may be a base table (or a view, see below) and a list of column names. The *privileges* include `SELECT`, allowing read access to the named columns of the indicated table, as well as `INSERT`, `UPDATE`, `DELETE`, with expected meanings. The *users* parameter may refer to a single user or a group of users. The `REVOKE` command is used to remove previously-granted privileges.

Once authenticated, clients can submit SQL statements to retrieve or update data. The database system parses the query and checks the client's permissions. If they are insufficient, an error message will be returned. The database administrator is often responsible for managing users in the system, maintaining the schema of the database, and defining initial access control policy. The confidentiality and integrity of stored data depends on the proper functioning of the access control mechanism and proper behavior of the administrator.

Views and access control

A view is a virtual table whose rows are determined by a view definition referencing stored tables (and possibly other views). A view definition is an expression in SQL that can com-

bine information from multiple tables, apply logical selection conditions, remove columns, compute aggregates, etc. Views can be used like base tables when querying the database, and in some cases can also be updated.

View creation and view access is controlled by the `GRANT` and `REVOKE` statements. Views add an important dimension to the access control mechanism in databases. For confidentiality, views can be defined to include precisely the data that should be permitted for a user, hiding other sensitive data items. Not all views can be unambiguously updated, so the SQL standard (as well as all commercial systems) restrict the updates that may be applied to views, independent of access controls. When views are updatable, the ability to perform updates can be mediated using `GRANT`, which can be a useful mechanism for ensuring data integrity.

While views increase the flexibility and power of the access control mechanism in databases, they also complicate policy definition and reasoning about what data is actually protected. We provide an example of these complications in Section 2.5, and study the issue in detail in Chapter 3.

Semi-structured data

Access control for semi-structured data is a less mature topic and lacks widely-accepted conventions. It typically offers fine-grained control over individual data items and requires some special features to cope with hierarchical data and uncertain structure.

Many languages [37, 69, 86, 108, 106, 68] have been proposed for describing access policies over XML documents. XML access control rules roughly follow the structure outlined above, specifying a *subject*, a set of *objects* (XML elements, attributes, or entire subtrees), an *effect* (usually grant or deny), and an *operation* (such as read, write, modify, or delete). A policy may consist of multiple rules with common or overlapping objects and it is therefore necessary to specify a conflict resolution policy for the rules. For example, rules denying access commonly take precedence over rules granting access. In addition, a default semantics usually denies access to all XML elements that are not the target of any rule.

Virtually all proposed access control languages use path languages derived from XPath

[30] to specify the target objects of a rule. An XPath expression returns a sets of target elements from the document. Some access control rules include features that specify *local* application (to the target element itself, and perhaps its attributes) or *recursive* application (to the target element and all its descendants). In Chapter 4 we describe our own model for expressing access control policies, which is based on well-established XML manipulation languages (XPath [30] and XQuery [18]) and we provide techniques for enforcing these policies using cryptography. Also, in Section 4.9 we review other work related to XML access control.

Access control in statistical databases

A statistical database is one which permits access only to views consisting of aggregate statistics for subsets of records in the database [3, 39]. Statistics include counts, sums, and averages, computed over subsets of records in a database. Much of the research on statistical databases has been motivated by the need to publish features of a population of individuals without revealing attributes of individuals. Note that the distinction between a conventional relational database and a statistical database is tenuous. Statistical databases are defined in terms of the type of data stored and the kind of queries permitted. Statistical data is very often stored in conventional relational database systems (along with other data that is not statistical in nature) and statistical queries can easily be defined using a relational query language.

Managing disclosure in such settings is a difficult problem because facts about individuals may be inferred or disclosed but with some uncertainty. Specialized techniques are required which are not the central focus of this dissertation. We do however return to disclosure analysis in statistical databases in Section 2.5.

2.3 Limitations of access controls

For our goal of securing distributed data exchange, access control mechanisms have two fundamental limitations, which constitute the founding motivations for the work of this dissertation. First, trusted enforcement of access control policies is required—a challenge

in a distributed setting because data owners do not control client systems. The goal of Chapters 4 and 5 is to relieve the need for trusted enforcement of access control by relying on cryptography. Second, controlling direct access to data objects is not always sufficient for protecting information. Despite a properly-enforced access control policy, unexpected disclosures may occur and inferences on the part of an adversary may be possible. In Chapter 3 we address the problem of unexpected disclosures by presenting a novel technique for analyzing the disclosure of basic relational views. We provide more background on the challenges of disclosure analysis shortly in Section 2.5. Below we review responses to the need for trusted enforcement.

The need for trusted enforcement

The advantages of distributed data exchange arise from the ability of peers in the system to provide storage and processing functions for other peers. Since peers are not trusted, we can only permit them to store data they themselves are permitted to access, threatening the utility of distributed data exchange. There are a two main ways to address the need for trusted enforcement in a distributed setting. The first relies on cryptography. The second makes remote systems trustworthy using special hardware.

Cryptography can be used to enforce access controls by storing data in encrypted form and disseminating keys to permit access. This has been referred to as a *passive protection mechanism* [72], as opposed to an active protection mechanism that has a trusted monitor determining access. This is the strategy followed in Chapter 4, using standard encryption so that data stored in encrypted form at a peer is not accessible and cannot be processed by that peer. It is worth noting that there are other encryption techniques that attempt to permit processing on encrypted data. These include homomorphic encryption [114, 49, 115] (which allows certain mathematical operations to be performed on encrypted data) and secure multiparty computation [140, 141, 119] (which allows a group of contributing parties to compute a function without revealing their input values to one another). Secure homomorphic encryption functions exist only for very limited operations, and secure multiparty computation has extremely high computational overhead for most applications. As a result,

we do not pursue complex processing of encrypted data in this dissertation.

A second approach to the problem posed by untrusted peers uses an architecture where a trusted computing base is installed at client sites. Since the data owner is not in physical control of these sites, some tamper-resistant hardware would be required. This has been proposed [22, 21, 20] in the context of data exchange using smartcards to provide very limited trusted resources at untrusted peers. The special hardware required makes this a significant departure from the goals of distributed data exchange, aimed at convenient and open access to data. We do not pursue this direction further.

A final response to the need for trusted enforcement shares some qualities of each of the above. It splits data across peers in such a way that collusion amongst some large number of peers is required in order for significant disclosures to take place. This strategy was proposed recently [4]. While promising, it still requires the combination of cryptographic techniques and a careful analysis of disclosure, two of the topics which are the subject of this dissertation.

2.4 Cryptographic Techniques

We review below basic concepts of cryptography relevant to Chapters 4 and 5, adopting Stinson's notation [126]. In addition, we provide a brief background on the security analysis of cryptographic algorithms and protocols. In particular, the notion of perfect secrecy, developed by Shannon for the analysis of cryptosystems, is relevant because it inspired our analysis of disclosure in Chapter 3.

2.4.1 Cryptographic primitives for confidentiality

A cryptosystem is a five-tuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ where $\mathcal{P}$ is a set of possible plaintexts, $\mathcal{C}$ is a set of possible ciphertexts, and $\mathcal{K}$ is the set of possible keys, the keyspace. $\mathcal{E}$ and $\mathcal{D}$ are sets of encryption and decryption functions, parameterized by keys. For each $k \in \mathcal{K}$, there is an encryption function $e_k \in \mathcal{E}$ and a corresponding decryption function $d_k \in \mathcal{D}$ such that $d_k(e_k(x)) = x$ for every plaintext $x \in \mathcal{P}$.

For confidentiality, it should be (i) computationally infeasible for an adversary to deter-

mine the decryption function d_k from observed ciphertext, even if the corresponding plaintext is known, and (ii) computationally infeasible to systematically determine the plaintext from observed ciphertext [39]. Examples of cryptosystems in common use are DES [56] and AES [59, 34].

Classical cryptosystems, are called *symmetric key cryptosystems* because d_k and e_k are either identical, or are easily derived from one another. These cryptosystems require prior secure transmission of the key before any enciphered communication begins. In a *public key cryptosystem* [48, 115], it should be computationally infeasible to derive d_k from e_k. In this case the encryption algorithm e_k is considered a public rule, which can be published freely, while the decryption algorithm is the private rule, kept secret.

2.4.2 Cryptographic primitives for integrity

Encryption does not by itself provide integrity. In public key cryptosystems, the encryption algorithm is available to anyone, and there is no protection against tampering with encrypted data. Even with symmetric key cryptosystems, where the encryption key is secret, integrity is not guaranteed. It is possible to modify certain blocks of ciphertext, without knowledge of the key, resulting in meaningful modification of the data when it is decrypted by the recipient. Integrity is ensured by using cryptographic hash functions and digital signatures.

Cryptographic hash functions

Cryptographic hash functions (also called message digests) are modeled after one-way functions, which are functions that can be efficiently computed but not inverted. A cryptographic hash function produces a short sequence of bytes when computed on its input. If the input is altered, the result of the hash function will change. For example, if h is a hash function, x a message and $y = h(x)$, then y can be considered an *authentication tag* [126]. Suppose y is stored securely, but x is vulnerable to tampering. When a message x' (purported to be x) is received, its hash $h(x')$ can be compared with y. If they are equal (and h is collision-free) then $x = x'$ and integrity is verified.

It may be the case that the message x is large, and made of constituent pieces (a file made

of blocks, or a set of data items). In this case, the technique above has the disadvantage that the entire message must be available for verification, and even small updates require re-computation of the hash function over the entire message. A Merkle hash tree [95, 96] addresses these limitations, defining an authenticated dictionary structure which permits efficient verification of elements of x and efficient updates to the message. Merkle trees are described in detail in Section 5.2. Their implementation in a relational database system is a main focus of Chapter 5.

A secure hash function should have the following properties. It should be computationally infeasible to find: a *preimage* (given y find x such that $h(x) = y$); a *second preimage* (given x, find x' such that $x' \neq x$ and $h(x') = h(x)$; a *collision* (given just h, find any x, x' such that $x \neq x$ and $h(x') = h(x)$. Common cryptographic hash functions have included MD5 [116], and the SHA [58] family of functions, however MD5 and SHA-1 have been broken in the recent past, and are now considered insecure. SHA-256 and other SHA variants are believed to be secure [113].

Digital signatures

On paper, signatures are intended as proof of authorship or an indication of agreement with the contents of a document. Paper signatures serve their purpose because they have some or all of the following properties [118]: the signature is unforgeable, the signature cannot be reused on another document, and the signed document is unalterable. A digital signature scheme is a pair of cryptographic algorithms: a protected signing algorithm applied to data to produce a short output string (the *signature*), and a public verification algorithm that is executed on the signed data and the signature to test authenticity.

Formally, a signature scheme is a five-tuple $(\mathcal{P}, \mathcal{A}, \mathcal{K}, \mathcal{S}, \mathcal{V})$ where $\mathcal{P}$ is a set of possible messages, $\mathcal{A}$ is a set of possible signatures, and $\mathcal{K}$ is again the keyspace. $\mathcal{S}$ and $\mathcal{V}$ are families of signing and verification algorithms, respectively, that are parameterized by keys. For each $k \in \mathcal{K}$, $\text{sign}_k : \mathcal{P} \rightarrow \mathcal{A}$ and $\text{verify}_k : \mathcal{P} \times \mathcal{A} \rightarrow \{\text{true,false}\}$. For every message $x \in \mathcal{P}$ and every signature $y \in \mathcal{A}$, $\text{verify}(x, y) = \text{true}$ if $y = \text{sign}(x)$ and $\text{verify}(x, y) = \text{false}$ if $y \neq \text{sign}(x)$.

The signing algorithm is private, and the verification algorithm is public. It therefore must be computationally infeasible to derive the signing rule, given the verification rule. It should also be computationally infeasible for an adversary to forge signatures, even given a set of signed messages. Common signature schemes include those based on the RSA [115] public key cryptosystem and the dedicated signature algorithm DSA [57].

2.4.3 Enforcing access control using cryptography

As we mentioned above, one of our goals is to provide access control in a distributed setting using cryptography. Gifford proposed the first such "passive protection mechanism" [72], designed to protect both confidentiality and integrity of data stored at a client, in the absence of a trusted protection system. In that work, cryptography is used to provide secrecy and integrity of data blocks or files, and access is granted by sharing an appropriate key set. The work presented in Chapters 4 and 5 shares the goal of ensuring confidentiality and integrity in an untrusted setting. It differs in its treatment of a very general data model protected at a fine granularity.

2.4.4 Security of cryptosystems

Shannon's notion of perfect secrecy [123] evaluates the information about the plaintext that is present in published ciphertext. Perfect secrecy, described below, inspired the definition of disclosure presented in Chapter 3 for information published using relational views.

The model for perfect secrecy of a cryptosystem assumes a probability distribution over the plaintext space $\mathcal{P}$. The plaintext message defines a random variable $\mathbf{x}$. The probability that a plaintext $x \in \mathcal{P}$ occurs is the *a priori* probability, $P[\mathbf{x} = x]$. The key $k \in \mathcal{K}$ is also chosen with a probability, $P[\mathbf{k} = k]$. These two probability distributions induce a probability distribution over the space of ciphertexts $\mathcal{C}$. The probability that a ciphertext $y \in \mathcal{C}$ will occur is denoted $P[\mathbf{y} = y]$, and can be easily computed from the *a priori* plaintext probabililities and the keyspace probabilities. A cryptosystem is perfectly secret if:

$$P[x] = P[x|y] \quad \text{for all } x \in \mathcal{P}, y \in \mathcal{C}$$

The term on the right is the conditional probability of the plaintext x occuring, given the

occurrence of a ciphertext y. Perfect secrecy asserts that the uncertainty about the plaintext message should be unchanged having seen the ciphertext, for all possible plaintexts and ciphertexts.

A simple example of a cryptosystem that is perfectly secure is the one-time pad. To encrypt a b-bit message x, a random b-bit key k is generated. Encryption and decryption consist of XOR with the key: $e_k(x) = x \oplus k$ and $d_k(y) = y \oplus k$. A new key must be generated for each plaintext encrypted. If keys are generated at random, and never reused, the ciphertext contains no information about the plaintext – perfect secrecy is satisfied.

For perfect secrecy to hold, the size of the keyspace $\mathcal{K}$ must be at least as large as the plaintext space $\mathcal{P}$ [122]. This means that the size of the key must be at least as large as the message encrypted. Since, in addition, keys cannot be reused, this is a major limitation, requiring that at least one bit of key material be communicated securely for each bit of encrypted data. Cryptosystems used in practice today are not perfectly secure. Further, it is impossible for a public-key cryptosystem to be unconditionally secure. This is because the encryption rule is public, so an adversary with unlimited resources can, given ciphertext y, always encrypt each plaintext in $\mathcal{P}$ until y is found [126].

There are many weaker notions of security studied by the cryptographic community [126]. For example, computational security measures the computational effort required to break a cryptosystem. It is very difficult to prove commonly-used cryptosystems computationally secure because it must be shown that the best algorithm for breaking the cryptosystem requires some large number of operations. Provable security does not provide an absolute guarantee of security, but instead asserts that breaking the cryptosystem is at least as hard as solving another well-studied problem believed to be computationally hard.

2.4.5 Protocol and application security

We discuss the security of encryption functions in the next section. It is worth noting here, however, that when encryption functions are used in protocols, or applied to collections of data items, proofs of security of the cryptosystems alone are not sufficient, as other disclosures may take place. There is a large body of work on the formal analysis of protocols

[93] which have been applied to authentication protocols, as in the BAN logic [27] (among others), and to data protection schemes, where the soundness of complex cryptographic expressions has been analyzed formally [1, 88, 9, 99, 100]. Protocol security is most relevant to the work presented in Chapter 4 where complex, nested encryption is applied to hierarchical data. Although the techniques of formal protocol analysis are beyond the scope of this thesis, Abadi and Warinschi have recently [2] provided a formal analysis of the protocol presented in Chapter 4. Further discussion is provided in Section 4.5.

2.5 Disclosure Analysis

Whenever data is derived from sensitive data and published, there is the possibility of disclosure. When the derived data is the result of applying encryption, disclosures may result from a weakness in the cryptographic function or from the way the function is applied to data. When the derived data is a view (statistical, or conventional) it may hide some sensitive data items, but it also may contain subtle clues about those hidden data items. The goals of disclosure analysis include classifying, measuring, and auditing disclosures, and may require modeling the prior knowledge of an adversary.

In this section we unify the analysis of disclosure as it is applied to cryptography and data publishing. We begin with basic concepts from disclosure analysis in databases. Then we present a series of examples, explaining informally some subtleties of disclosure. In the last part of this section we relate the notions of perfect secrecy and entropy to disclosure in databases.

Disclosure in databases

It is common practice in databases to study disclosure with respect to some sensitive property or query Q. (Naturally, this does not lose generality since Q could be the identity query on the database). An exact disclosure occurs when the answer to Q is revealed with certainty. An approximate disclosure occurs when Q is not determined exactly, but information is nevertheless revealed. Three types of approximate disclosure were identified by Dalenius [35] in the context of statistical databases. A disclosure may reveal bounds of

an ordered attribute, for example that the answer to query Q lies in the range $[l_1, l_2]$. A probabilistic disclosure reveals that the answer to Q is in the range $[l_1, l_2]$, but only with some probability p, called the confidence. Finally, a negative disclosure reveals that the answer to Q is not value x.

The adversary's knowledge

A user's starting knowledge, sometimes called *a priori* knowledge, or *working* knowledge [39], consists of facts known by the user prior to any interaction with the database. For example, a user may know, *a priori*, the set of possible medical conditions that might be contained in the database. Such knowledge is often assumed to include the schema of the database itself as well as common sense facts about the domain. Such prior knowledge is information that is not contained in the database, but may be acquired through external sources or simply assumed as fact. In the model of perfect secrecy, the adversary's prior knowledge about the plaintext is represented by the *a priori* probability distribution.

2.5.1 Examples of disclosure of protected data

We now present a series of examples illustrating disclosures that can result from protecting a relational table with cryptographic techniques, conventional access control mechanisms, and statistical protection mechanisms. The Employee table shown in Table 2.1 contains sensitive information about named employees such as contact details, salary, and marital status. Typically, it is difficult to hide the schema of a protected table—it is often easy for an adversary to predict and does not change often. The emphasis is on protecting the records in the database.

Example 2.5.1 Block encryption

Table 2.5.1 contains the ciphertext resulting from the encryption of the Employee table (represented as tab-delimited text) using the AES [59] encryption algorithm and an 128-bit key derived from the password `secretsecret`. The ciphertext bytes were converted to

Table 2.1: The Employee table

Name	Dept	Phone	Salary	Marital Status
Edward	Management	584-2154	80,000	Married
Elaine	Accounting	584-2845	60,000	Single
Fiadora	Admin	584-7515	35,000	Married
Frank	HR	584-6324	45,000	Divorced
Gerard	Accounting	584-9521	55,000	Single
Jane	Management	584-2834	85,000	Married
Joe	Admin	584-1147	35,000	Married

Table 2.2: The Employee table, under block encryption.

```
U2FsdGVkX1/OlG/JmkMv+U4r//F+Z7BCIo54Vv5ApI90cdTkK4YCxBgPwkQ0JgV6
5N+N9kBOelPNuilzcJ9oaRtp09SBVqXOvF+wUc2GTocKdDA893pj4PTS8WE5grog
dGUGyEHJW7ZbBIQ/ivxBG4ZZK5i9cwFgH7agLpF8u4KmTwjWiJU/jrawf1mVWE2A
QZADFn3FNLOOO7YOz9OO5lIIFT5xUy9QfqlIFtJQ9IPywvZMYPCOFWR65J8dDKkv
vFapShcralzrXUx5s4bryFVigPUrE5i9Wi8kxDa4zpKhAcTetaaPgHqPlQTyni3c
hx5pd3zXd+qsWWXJPXkpHSnHzJYqdQOxsvKEw3+TKZHtp0GjITF+FqC1PGrGGSL0
AfzknwOItWVlXJokRU2zhJOmQ28H3Td2tVJ+PKeHvNd5z784/RJTZ2h70Tl84+t6
qTkjgSFiA4y7XwDjmWTgcKC7Ude/8mFt9yc5gBdO1xtSk88CpyA4xdMirrm4db0h
1iYx/DuQIJLpvpZX0bdFYqjgUwfqV9l5n/0WTZ1yYG0=
```

base-64 text for presentation purposes.

Practical encryption functions like AES are not perfectly secret. This is immediate, since the plaintext space (here, all possible Employee instances, constructed from fixed attribute domains) is much larger than the key space (2^{128}). The ciphertext therefore contains information about the input table. In particular, an adversary with unlimited resources could discover that some Employee instances would never produce this ciphertext output, under any key. That information is extremely difficult to compute for computationally bounded adversaries. Even so, the ciphertext is likely to reveal the size of the encrypted table, to a close approximation. For example, an adversary may be able to estimate, from knowledge of the schema, the average size in bytes of a record occurring in the Employee table.

An adversary could then reason about the number of employees in the Employee database, obscured only by padding of the plaintext prior to encryption.

Example 2.5.2 Attribute-value encryption

The Employee table, encrypted attribute-by-attribute, is pictured in Table 2.3. While individual attribute values are hidden by encryption, the exact number of rows in the table is revealed. Further, if the same encryption key is used for each column of values (as in the example), duplicates values are revealed. An adversary may learn, for instance, that there are 4 distinct departments, and that there are two employees in the same department with the same salary. There are ways to encrypt individual attributes without revealing duplicates, but there are also advantages to allowing these disclosures. In fact, a number of recent proposals in the literature [79, 78, 5] publish tables of data encrypted in this way, so the analysis of disclosure remains of interest. Prior knowledge or statistical expectations about the data allow further inferences (see Example 2.5.5 below).

Table 2.3: The Employee table under attribute-value encryption. A few of the duplicate values are underlined.

Name	Dept	Phone	Salary	Marital Status
b77f288091	f1ab2b289e	212b030658	bfced00a5a	46363ce136
a03d85bfc5	a4bf0a4be2	0d4f0044d1	c96fc1bc4a	36a4fa81f5
a4117cb167	35a1800c53	8f9a3edcc8	f886ffc571	46363ce136
fde1f029e2	81a5ca034a	32d54ab559	af924dcfc5	1211bdff44
6feedd319b	a4bf0a4be2	f9422b55a6	4edce9bc47	36a4fa81f5
e34529bcc2	f1ab2b289e	94159bbf91	a048e9dcc7	46363ce136
0f555d4ddc	35a1800c53	d2cc54a35d	f886ffc571	46363ce136

Example 2.5.3 Basic relational views

The Personnel department may wish to publish two views of the Employee table for legitimate use by others. For example, the view projecting on (*Name*, *Department*), and the view projecting (*Department*, *Phone*), as shown in Table 2.4. (Note that there is no encryption here.) While the Personnel department may be willing to share these two views, they may

at the same time wish to keep employees' phone numbers secret. This amounts to protecting the following query:

$$Q = \Pi_{Name,Phone}(\mathsf{Employee})$$

Although the exact answer to Q cannot be computed from the published views (since the association between *name* and *phone* is not present) their combination contains partial information about names and phone numbers of employees. For instance, since Edward works in the Management department, the adversary knows his phone number is either 584-2154 or 584-2834.

Table 2.4: Two relational views derived from the Employee table. (Left) $V_1 = \Pi_{name,dept}(\mathsf{Employee})$ and (Right) $V_2 = \Pi_{dept,phone}(\mathsf{Employee})$.

Name	Dept
Edward	Management
Elaine	Accounting
Fiadora	Admin
Frank	HR
Gerard	Accounting
Jane	Management
Joe	Admin

Dept	Phone
Management	584-2154
Accounting	584-2845
Admin	584-7515
HR	584-6324
Accounting	584-9521
Management	584-2834
Admin	584-1147

Example 2.5.4 Statistical views

Table 2.5(left) shows a simple statistical view of the database consisting of the average employee salary for each department. This data obviously offers an adversary approximate information about individual salaries. More severe disclosures result from combining this aggregate data with prior knowledge (see below), when individual salaries take on maximum or minimum values, or when an adversary can witness multiple versions of the database published over time. Table 2.5(right) shows the same view, but after random perturbation of values.

Example 2.5.5 Prior knowledge

Prior knowledge refers to the facts or relationships an adversary may come to know from

Table 2.5: (Left) Average salary by department. (Right) Average salary by department, protected by randomization.

Dept	Avg Salary
Accounting	57,500
Admin	35,000
HR	45,000
Management	82,500

Dept	Avg Salary
Accounting	58,123
Admin	34,435
HR	46,712
Management	80,892

other sources that are outside the control of the data owner. These can be the basis of further inferences, and can impact the disclosure in any of the examples above. For example, an adversary may know that `Married` is the most common value for marital status amongst employees. In Example 2.5.2, this allows an adversary to infer that the two employees in the same department with the same salary are both married. If it is known that there is only one employee in the `HR` department, publishing the average salary by department, as in Example 2.5.4, causes a severe disclosure, diminished only slightly by randomization. In general, prior knowledge may be derived from other data sources, from known correlations between data values, or from the knowledge that constraints hold over the protected data. Modeling the prior knowledge that may be available to an adversary is a critical but extremely difficult aspect of disclosure analysis.

2.5.2 Analyzing and quantifying disclosure in databases

Shannon's theory of information [123] can be used to measure the security of data protected by views. Entropy is a measure of the amount of information in a message. Entropy is determined by the probability distribution over the space of all possible messages. If $\mathcal{P}$ is the space of plaintext messages, and a message $x \in \mathcal{P}$ occurs with probability $p(x)$, then the entropy of a message is:

$$-\sum_{x \in \mathcal{P}} p(x) \log_2 p(x)$$

This quantity is equal to the expected number of bits in optimally encoded messages. It is also a measure of the uncertainty about a message.

Entropy can be used as a measure of disclosure by studying the entropy of the sensitive

query Q. A probability distribution over databases induces a probability for each possible answer to Q. These are the *a priori* probabilities, from which the prior entropy can be computed. The publication of a view of the database induces *posterior* probabilities, changing the probabilities of answers to Q, and perhaps making some answers impossible. If the posterior entropy of Q is zero, then an exact disclosure has occurred. Otherwise, the difference in prior and posterior entropies provides a measure of disclosure. Denning notes that requiring identical prior and posterior entropies is too strong a condition for evaluating statistical queries – no statistics could be published under this condition [39].

Perfect secrecy (described in Section 2.4) can be applied to relational views. It depends on prior and posterior probabilities of answers, as above, but results in a stronger condition than one based on entropy. The following example illustrates this fact:

Example 2.5.6 Entropy and Perfect Secrecy compared
Consider a boolean query Q, returning true or false on database instances. Suppose the *a priori* probabilities for Q are:

$$P[Q = \text{true}] = 1/10 \text{ and } P[Q = \text{false}] = 9/10$$

Further, assume that as the result of publishing some view V, the probabilities for answers to Q change to:

$$P[Q = \text{true}|V] = 9/10 \text{ and } P[Q = \text{false}|V] = 1/10$$

The prior entropy of Q and posterior entropy of Q are equal, indicating zero disclosure. But intuitively a substantial approximate disclosure has in fact taken place: as a result of seeing the view, an adversary's expectation of the answer to Q changes substantially. Perfect secrecy captures this disclosure because it requires that $P[Q] = P[Q|V]$ for all possible answers, and thus fails for this example.

Another measure of disclosure was described recently by Evfimievski, et al [54]. Published data V is said to cause a ρ_1-to-ρ_2 privacy breach for a property Q if the prior probability is small (less than or equal to parameter ρ_1) and the posterior probability is large (greater than or equal to parameter ρ_2). The query in Example 2.5.6 constitutes a

10%-to-90% privacy breach. The goal is to choose ρ_1 and ρ_2 to capture properties that were very unlikely prior to releasing V, and have become likely as a result of the release. It is not clear how to choose these parameters in general, and whether they are dependent on the property Q or on the application.

There is no general framework for analyzing disclosure in the range of cases illustrated above, and many open problems remain. One important problem is addressed in Chapter 3 where we provide a formal analysis of the disclosure of simple relational views (as in Example 2.5.3), using a definition of security related to perfect secrecy. Prior to that work, no formal tools existed for precisely analyzing disclosure.

Chapter 3

MANAGING DISCLOSURE

3.1 Introduction

In this chapter we provide a theoretical analysis of information disclosure resulting from the publication of relational views. Specifically, we study the following fundamental problem, called the *query-view security* problem: given views $V_1, V_2, \ldots$ that we want to publish, do they logically disclose any information about a query S that we want to keep secret? The views are expressed in a query language, and may remove data items through projections or selections, or break associations between data items. The query expressions $S, V_1, V_2, \ldots$ are known by the adversary, the answers to $V_1, V_2, \ldots$ are published, while the underlying database remains secret.

A spectrum of information disclosure

To motivate the problem, we describe through examples the variety of disclosures that may result from the publication of relational views. Table 3.1 contains a set of query-view pairs referring to an Employee relation, along with an informal description of the information the views disclose about the query. For simplicity, we use an abbreviated version of the Employee table from Section 2 consisting only of *name*, *department*, and *phone* attributes. The examples represent a spectrum of information disclosure, beginning with total disclosure and ending with a secure query and view.

The first query and view is an obvious example of a total disclosure because S_1 is answerable using V_1. Example (2) is precisely the Example 2.5.3 of Section 2, where two views are published, and the relationship between *name* and *phone* is sensitive. As we noted, for small departments, it may be easy for the adversary to guess the association between names and phone numbers.

As another example of partial disclosure, consider example (3), whose view is the projection on the *name* attribute: $V_3 = \Pi_{name}(\mathsf{Employee})$. We ask whether query $S_3 =$

Table 3.1: Pairs of views and queries, over relation $\mathsf{Employee}(name, department, phone)$ and an informal description of their information disclosure.

	View(s)	Query	Information Disclosure	Query-View Security
(1)	$V_1(n,d) : -\mathsf{Emp}(n,d,p)$	$S_1(d) : -\mathsf{Emp}(n,d,p)$	Total	No
(2)	$V_2(n,d) : -\mathsf{Emp}(n,d,p)$ $V_2'(d,p) : -\mathsf{Emp}(n,d,p)$	$S_2(n,p) : -\mathsf{Emp}(n,d,p)$	Partial	No
(3)	$V_3(n) : -\mathsf{Emp}(n,d,p)$	$S_3(p) : -\mathsf{Emp}(n,d,p)$	Minute	No
(4)	$V_4(n) : -\mathsf{Emp}(n,\mathsf{Mgmt},p)$	$S_4(n) : -\mathsf{Emp}(n,\mathsf{Admin},p)$	None	Yes

$\Pi_{phone}(\mathsf{Employee})$ is secure when this view is published. In this case the view omits phone entirely and would seem to reveal nothing about phone numbers in S_3. Surprisingly, the view does disclose some information about the secret query. In particular, it can reveal something about the size of the Employee relation, and therefore contains some small amount of information about the omitted column. We describe this further in Section 3.3.

The last example, Table 3.1(4), is a case where no information is disclosed. The names of employees in the Management department reveal nothing about the names of employees in the Admin department.

Known techniques

Classical techniques for analyzing the relationship between queries and views fail in the context of information security. For example, the basic problem addressed in query answering [80] is: given a view V (or several such views), answer a query S by using only the data in the view(s). A more refined version, called query rewriting, asks for the answer to be given as a query over the view(s) V. Whenever S can be answered from V, then S is obviously not secure, as in Table 3.1(1). However, adopting non-answerability as a criterion for security would clearly be a mistake: it would classify example (2) as secure. As we claim

above, even though the query S may be not answerable using V, substantial information about the query may be revealed, allowing an attacker to guess the secret information with a high probability of success.

A related strategy considers (for a database instance I and view V) the view answer $v = V(I)$ as a constraint on the set of possible database instances, making some impossible. The set of possible answers to a query S given V may therefore be reduced. (If this set happens to have size 1, then the answer to S is determined by $v = V(I)$.) We might say S is secure given V if *every* possible answer to S remains possible given V. This criterion would classify Examples (2) and (3) correctly. That is, it would capture the partial disclosures in these cases, but not in others. However, this standard of security ignores the *likelihood* of the possible answers of S. For example, consider the boolean query and view below:

$$S() : -\mathsf{Employee}(\mathsf{Jane}, \mathsf{Shipping}, 1234567)$$

$$V() : -\mathsf{Employee}(\mathsf{Jane}, \mathsf{Shipping}, p), \mathsf{Employee}(n, \mathsf{Shipping}, 1234567)$$

In the absence of the view, the query S (which asserts the presence of a particular tuple in the database) may be true or false. Given the answer to V on the database, S could still evaluate to true or to false. However, the probability that S is true is substantially higher given that V is true, and so a serious disclosure has occurred. In general, while V may not rule out any possible answers to S, some answers may become less likely (or in the extreme, virtually improbable) without contradicting a security criterion based on possible answers. The definition of query-view security studied here captures this disclosure.

Contributions

The first contribution is a formal definition of query-view security that captures the disclosure of partial information. Inspired by Shannon's notion of *perfect secrecy* [122], the definition compares the likelihood of an adversary guessing the answer to secret query S with and without knowledge of views $V_1, V_2, \ldots V_n$. When the difference is zero, we say that the query is secure w.r.t. the views. To the best of our knowledge this is the first attempt to formalize logical information disclosure in databases. The second contribution consists of a number of theoretical results about query-view security: we prove a necessary and

sufficient condition for query-view security, and show that the security problem for conjunctive queries is Π_2^p-complete; we generalize query-view security to account for pre-existing knowledge; and when the query is not secure with respect to a view, we characterize the magnitude of disclosure.

Chapter Organization

The next section presents notation and a probabilistic model of databases. Section 3.3 describes our definition of query-view security and its main results. Section 3.4 extends these results to include prior knowledge, and in Section 3.5 we discuss attempts to weaken the definition of disclosure. We treat encrypted views in Section 3.6 and related work in Section 3.7.

3.2 Background and Notation

As we mentioned, many disclosures do not involve an adversary computing S completely according to standard database query semantics. Instead a partial disclosure reveals to the adversary something about the likelihood of answers to a secret query S. After an overview of notation, we present our security model that allows formal statements about the probability of a database and query answer.

3.2.1 Basic Notation

We assume a standard relational schema consisting of several relation names $R_1, R_2, \ldots$, each with a set of attribute names. Let D be the finite domain, which includes all values that can occur in any attributes in any of the relations. For example D may be the set of decimal numbers up to an upper bound, and all strings up to a given length. In a particular setting we may consider further restricting D, e.g. to include only valid disease names, valid people names, or valid phone numbers.

We use datalog notation to denote tuples belonging to the relations of the given schema. For example $R_1(a, b, c)$ denotes a tuple in R_1, and $R_3(b, a, a)$ denotes a tuple in R_3. Let tuples(D) be the set of all tuples over all relations in the schema that can be formed with constants from the domain D. A *database instance* I is any subset of tuples(D), and we

denote by *inst*(D) the set of all database instances over the domain D. A *query* of arity k is a function Q : *inst*(D) $\rightarrow \mathcal{P}(D^k)$. For an instance I, $Q(I)$ denotes the result of applying Q to I. A boolean query is a query of arity 0. A *monotone* query has the property $I \subseteq I' \Rightarrow Q(I) \subseteq Q(I')$. In most of the paper our discussion will focus on *conjunctive queries with inequalities*, written in datalog notation. For example:

$$Q(x) : -R_1(x, a, y), R_2(y, b, c), R_3(x, -, -), x < y, y \neq c$$

Here x, y are variables, $-$ are anonymous variables (each occurrence of $-$ is distinct from all others) and a, b, c are constants.

3.2.2 The Security Model

We assume a probability distribution on the tuples, Pr : tuples(D) $\rightarrow [0, 1]$, s.t. for each $t_i \in$ tuples(D), $\Pr(t_i) = x_i$ represents the probability that the tuple t_i will occur in a database instance. We will refer to the pair $(D, \Pr)$ as a *dictionary*. A dictionary induces a probability distribution on specific instances: for any $I \in$ *inst*(D), the probability that the database instance is precisely I is:

$$\Pr[I] \quad = \quad \prod_{t_i \in I} x_i \cdot \prod_{t_j \notin I} (1 - x_j) \tag{3.1}$$

For example, if the schema consists of a single table Patient(*name, disease*) representing sensitive data in a hospital, then the domain D may consist of all possible names (e.g. those occurring in a phone book for the entire country), together with all possible diseases cataloged by the CDC. For each tuple t_i=Patient(*name*, *disease*), $\Pr(t_i)$ is the (very small) probability that a person with that name and that disease is in the hospital's database. To illustrate, assuming 10^8 distinct names and 500 distinct diseases[1] there are $n = 5 \times 10^{10}$ tuples in tuples(D), and one possible probability distribution is $\Pr(t_i) = 200/n$ for every $t_i \in$ tuples(D). This is a uniform probability distribution, for which the expected database size is 200 tuples. A more accurate, but far more complex probability distribution is one that takes into account the different risk factors of various ethnic groups and for each diseases. For example the probability of a tuple Patient("John Johnson","Cardiovascular Disease")

[1]Fewer than 500 are listed at `http://www.cdc.gov/health/`.

will be slightly higher than the probability of the tuple Patient("Chen Li","Cardiovascular Disease"), if Americans have a higher risk of a Cardiovascular Disease than Chinese, and the nationality of John Johnson is likely to be American while that of Chen Li is likely to be Chinese.

The probability $\Pr(t_i)$ may be too complex to compute in practice, but computing it is not our goal. Instead we will *assume* that the adversary can compute it, and can use it to derive information about the secret query S. Thus, we endow the adversary with considerable power, and study under which circumstances no information is disclosed.

Given a probability distribution over database instances, a query S attains some answer s with probability equal to the sum of the probabilities of the satisfying instances:

$$\Pr[S(I) = s] \quad = \sum_{\{I \in inst(D) | S(I)=s\}} \Pr[I] \tag{3.2}$$

Note that in the model studied here, occurrences of tuples in the database are independent probabilistic events. This is a limitation. In practice, the occurrence of tuples may be correlated due to underlying relationships in the data or integrity constraints. If tuples are positively correlated (respectively, negatively correlated) the presence of one tuple increases (decreases) the likelihood of another. For example, a key constraint introduces strong negative correlations. We will address some of these limitations in Section 3.4 by studying query-view security relative to *prior knowledge* expressing a functional dependency. However, extending our results to a model that can capture arbitrary correlations between tuples remains an open problem.

3.3 Query-View Security

In this section we formalize our notion of query-view security, describe its basic properties, and state our main theorems which result in a decision procedure for query-view security.

3.3.1 Definition of Query-View Security

Our standard for query-view security is inspired by Shannon's definition of *perfect secrecy* [122]. Let $\bar{V} = V_1, \ldots, V_k$ be a set of views, and S a "secret" query. Both the views and the query are computed over an instance I of a relational schema. We consider

an adversary who is aware of the domain and probability distribution over instances (the dictionary), and is given $\bar{V}(I)$ (but not I). The adversary's objective is to compute $S(I)$. The definition below captures the intuition that $\bar{V}(I)$ discloses no information about $S(I)$. Below, $\bar{V}(I) = \bar{v}$ means $V_1(I) = v_1 \wedge \ldots \wedge V_k(I) = v_k$.

Definition 3.3.1 (Query-View Security) *Let (D, Pr) be a dictionary. A query S is secure w.r.t. a set of views $\bar{V}$ if for any possible answer s to the query, and any possible answers $\bar{v}$ to the views, the following holds:*

$$Pr[S(I) = s] = Pr[S(I) = s \mid \bar{V}(I) = \bar{v}] \tag{3.3}$$

Query-view security is denoted $S \mid_{Pr} V$, or simply $S \mid V$ if Pr is understood from the context.

The left hand side of equation (3.3) represents the *a priori* probability that S attains a particular answer s over the instance I, which can be computed by the adversary using $(D, \Pr)$. The right hand side is also the probability that $S(I) = s$ but conditioned on the fact that $\bar{V}(I) = \bar{v}$. The security condition asserts the equality of these two probabilities (for all possible $s, \bar{v}$) and therefore says that nothing beyond the *a priori* knowledge is provided by $\bar{V}$. Equation (3.3) is also the familiar definition of independence of two statistical events. Accordingly, S is secure w.r.t. $\bar{V}$ iff S and $\bar{V}$ are statistically independent events. We can rewrite (3.3) as follows:

$$\Pr[S(I) = s]\Pr[\bar{V}(I) = \bar{v}] = \Pr[S(I) = s \wedge \bar{V}(I) = \bar{v}] \tag{3.4}$$

Next we apply the definition in two examples:

Example 3.3.2 Non-security Consider a single relation $R(X, Y)$ and domain $D = \{a, b\}$. There are 4 possible tuples $R(a, a), R(a, b), R(b, a), R(b, b)$, and the set of instances $inst(D)$ contains the 16 subsets of these. Assume for simplicity that $\Pr(t_i) = 1/2$ for each tuple t_i, and consider the following query and view:

$$V(x) : -R(x, y)$$
$$S(y) : -R(x, y)$$

V projects the first attribute of R while S projects the second. Although we might expect that the view provides no information about the query, it is actually not the case that $S \mid V$.

Informally, the answer to V contains some information about the size of the database which impacts answers to S. Consider a particular answer $\{(a)\}$ for S. There are 3 equally-likely instances generating this answer: $\{R(a,a)\}$, $\{R(b,a)\}$, and $\{R(a,a), R(b,a)\}$. Therefore, we have *a priori* probability:

$$\Pr[S(I) = \{(a)\}] = 3/16$$

Now suppose we are given that $V(I) = \{(b)\}$. There are again 3 instances, only one of which causes $S(I) = \{(a)\}$. So, because each instance is equally-likely we have:

$$\Pr[S(I) = \{(a)\} \mid V(I) = \{(b)\}] = 1/3$$

This contradicts (3.3) for the particular answers considered, and it follows that S and V are *not* secure for this particular probability distribution. We show in the next section that they are not secure for any distribution.

Example 3.3.3 Security As an example of a secure query and view, consider the same schema and dictionary, and:

$$V(x) :- R(x,b)$$

$$S(y) :- R(y,a)$$

Here S is secure w.r.t. V. We prove this later, but illustrate here with one example. Consider one possible output of S: $S(I) = \{(a)\}$. There are 4 instances that lead to this output, $\{R(a,a)\}$, $\{R(a,a), R(a,b)\}$, $\{R(a,a),\ R(b,b)\}$, and $\{R(a,a),\ R(a,b),\ R(b,b)\}$, hence:

$$\Pr[S(I) = \{(a)\}] = 4/16 = 1/4$$

Consider also one possible output of V, say $V(I) = \{(b)\}$. There are four instances I satisfying this constraint: $\{R(b,b)\}$, $\{R(b,b),\ R(a,a)\}$, $\{R(b,b), R(b,a)\}$, $\{R(b,b),\ R(a,a),\ R(b,a)\}$. Of these only one also results in $S(I) = \{(a)\}$, hence:

$$\Pr[S(I) = \{(a)\} \mid V(I) = \{(b)\}] = 1/4$$

One can manually check, for all possible combinations of outputs of S and V, that the probability of S is unchanged by publishing V. We will provide an easier criterion for checking this shortly.

3.3.2 Properties of query-view security

Several properties of query-view security follow, providing intuition and justifying our choice of definition.

Reflexivity It follows from Bayes' Theorem that security is a reflexive relation: $S \mid \bar{V}$ iff $\bar{V} \mid S$.

Security (not obscurity) We always assume that publishing the views $\bar{V}$ includes exposing both the view definitions and their answers over the hidden database. Basing the security on concealing the view and query expressions is dangerous. We thus avoid the pitfall of "security by obscurity", identified long ago by the cryptographic community as ineffective [130, 118].

Instance-independence If the query S is secure w.r.t. the views $\bar{V}$, it remains so even if the underlying database instance I changes: this follows from the fact that Eq.(3.3) must hold for any query output s and any view outputs $\bar{v}$. We say that query-view security is *instance independent*. This property is necessary in applications like message-based data exchange, where messages are exchanged continuously, even as the database changes. Once $S \mid \bar{V}$ has been checked, the views $\bar{V}(I)$ can safely be exchanged without any compromise of $S(I)$. In fact, one can prove that if successive instances are independent from one another, then even the adversary collects snapshots of the views at various moments of time, $\bar{V}(I_1)$, $\bar{V}(I_2), \ldots$, $\bar{V}(I_t)$, he still cannot learn anything about any of $S(I_1), \ldots, S(I_t)$. This way of defining security is different from the standard definition in statistical databases. There the security criteria often apply to a particular database instance, and may fail if the instance is later updated. For example, one security criterion requires that the aggregate function be computed only on cells that are "large enough". One data instance may be secure, but it becomes insecure when tuples are deleted (making some cells too small), or when tuples are inserted (creating new cells, which are small).

Dictionary-independence The definition of query-view security $S \mid \bar{V}$ is for a particular

dictionary $(D, \Pr)$. In practice, however, the dictionary is often ill-defined: for example the probability distribution Pr is impossible to compute, and even the domain D may be hard to define precisely. Thus, we are interested in a stronger version of security, which is *dictionary-independent*. Our results in the next section provide necessary and sufficient conditions for dictionary-independent security. They show, surprisingly, that, in some cases, security for *some* dictionary implies security for *all* dictionaries (see Theorem 3.3.8 and Proposition 3.3.9).

Collusions Given $\bar{V} = V_1, \ldots\ V_k$, we will show in Theorem 3.3.5 that $S \mid \bar{V}$ if and only if $S | V_i$ for all $i = 1, \ldots, k$. This has the following consequence. If we send different views to different users, but have determined that the secret query S is secure w.r.t. each view separately, then nothing will be leaked about S even if the recipients collude, i.e. exchange the views they received and try to learn something about S by examining all the views together. This strong property is a consequence of our adoption of a notion of *perfect secrecy* to define security. Disclosure through collusion happens when each view leaks very little information when taken separately, but together may leak a lot of information about S. We will re-examine collusion in Sec. 3.5 when we discuss measuring disclosures.

Query answering The database community has studied extensively the following *query answering* problem. Given a set of views $\bar{V} = V_1, \ldots\ V_k$ and another view V' find a function f s.t. for any instance I, $V'(I) = f(\bar{V}(I))$: in this case we say that V' is answerable from $\bar{V}$. In the related *query rewriting* problem, f is restricted to be expressed in a query language. It is natural to ask about the relationship to security. Intuitively, if V' is answerable from $\bar{V}$, then the information content of V' is not more than that of $\bar{V}$, and any query S which is secure w.r.t. $\bar{V}$ should be also secure w.r.t. to V'. This intuition is correct, as the following straighforward argument shows. We have

$$\Pr[V'(I) = v'] = \sum_{\bar{v}} \{\Pr[\bar{V}(I) = \bar{v}] \mid f(\bar{v}) = v'\}$$

and

$$\Pr[S(I)=s \wedge V'(I)=v'] = \sum_{\bar{v}} \{\Pr[S(I)=s \wedge \bar{V}(I)=\bar{v}] \mid f(\bar{v})=v'\}$$

which implies that the view V' satisfies Equation (3.4). In particular, if V is a boolean view, then it follows that $S \mid V$ iff $S \mid \neg V$. A similar result holds when security fails: if $\neg(S \mid \bar{V})$ and another query S' is computable from S, then $\neg(S' \mid \bar{V})$.

Aggregates When applied to queries with aggregates our definition of security results in a very strict condition: no query and view containing an aggregate over a common tuple are secure. Techniques from statistical databases are better-suited for the case of queries with aggregates, and are orthogonal to our discussion. We therefore omit aggregate functions from the query language we consider.

3.3.3 Fundamental Theorems of Query-View Security

At this point, the only obvious procedure for deciding query-view security is to compute probabilities for each answer to the query and view. In addition to the computational complexity of this strategy, it requires re-computation for each dictionary. In this subsection we present techniques for deciding query-view security by analyzing the query and view definitions, and prove that this technique is dictionary-independent.

Definition 3.3.4 (Critical tuple) *Let D be a finite domain and Q be a query. A tuple $t \in tuples(D)$ is critical for Q if there exists an instance $I \in inst(D)$ such that $Q(I - \{t\}) \neq Q(I)$. The set of critical tuples of Q is denoted $crit_D(Q)$, or simply $crit(Q)$ when D is understood from the context.*

The intuition is that t is critical for Q if there exists some instance where dropping t makes a difference.

For a simple illustration, consider the boolean query $Q() : -R(a_1, x)$ and let $D = \{a_1, \ldots, a_n\}$. Any tuple of the form $R(a_1, a_i)$, $i = 1, \ldots, n$, is critical for Q, because Q returns true on the database consisting of the single tuple $R(a_1, a_i)$, but if we remove that tuple then we get the empty database on which the query returns false.

We can now formulate the characterization of query-view security. The proof is in Sec. 3.3.4.

Theorem 3.3.5 *Let D be a domain. Let S be a query and $\bar{V}$ be a set of views. Then $S \mid_{Pr} \bar{V}$ for every probability distribution Pr iff $crit_D(S) \cap crit_D(\bar{V}) = \emptyset$.*

Here $crit(\bar{V})$ is $crit(V_1) \cup \ldots \cup crit(V_k)$. In particular it follows that $S \mid_{\mathrm{Pr}} \bar{V}$ forall Pr iff $S \mid_{\mathrm{Pr}} V_i$ forall $i = 1, \ldots, k$ and forall Pr. The theorem says that the only way a query can be insecure w.r.t. some views is if they have some common critical tuple. This result translates the probabilistic definition of query-view security into a purely logical statement, which does not involve probabilities. This is important, because it allows us to reason about query-view security by using traditional techniques from database theory and finite model theory.

Next we revisit the query and view examples from the last section and apply Theorem 3.3.5.

Example 3.3.6 In Example 3.3.2, we saw that security fails to hold for $V(x) : -R(x, y)$ and $S(y) : -R(x, y)$. Every tuple is critical for V: for example, $R(a, b)$ is critical for V because $V(\{\}) = \{\}$ while $V(\{R(a, b)\}) = \{(a)\}$. Similarly, every tuple is critical for S, so because $crit(V) \cap crit(S)$ is nonempty, we conclude $\neg(S \mid_{\mathrm{Pr}} V)$ at least for some probability distribution Pr.

Example 3.3.7 We argued in Example 3.3.3 that security holds for $V(x) : -R(x, b)$ and $S(y) : -R(y, a)$. The critical tuples of S are $crit(S) = \{R(a, a), R(b, a)\}$, and similarly $crit(V) = \{R(a, b), R(b, b)\}$. Because $crit(S) \cap crit(V) = \emptyset$, Theorem 3.3.5 allows us to conclude $S \mid_{\mathrm{Pr}} V$ for every probability distribution Pr.

So far S and $\bar{V}$ were allowed to be arbitrary queries. We now restrict S and $\bar{V}$ to be monotone queries and will prove that the definition of query-view security is, for all practical purposes, dictionary-independent. The main step is the following theorem, whose proof is in Sec. 3.3.4.

Theorem 3.3.8 (Probability-Independence) *Let D be a domain, and $S, \bar{V}$ be any monotone queries. Let Pr_0 be a probability distribution s.t. $\forall t, Pr_0(t) \neq 0$ and $Pr_0(t) \neq 1$. If $S \mid_{Pr_0} \bar{V}$ then for every probability distribution Pr, $S \mid_{Pr} \bar{V}$.*

This is a surprising theoretical result, which says that if a query is secure even for one probability distribution, then it is secure for all such distributions. Continuing Example 3.3.2, both S and V are monotone. It follows that $\neg(S \mid_{\Pr} V)$ for any probability distribution Pr which is $\neq 0$ and $\neq 1$. Notice that for the trivial distribution $\Pr(t) = 1$, $\forall t$, we have $S \mid_{\Pr} V$, because in this case the answer to both S and V are known.

We still need to show that the definition is insensitive to a particular choice of domain, and for that we will further restrict all queries to be conjunctive queries. As we vary the domain D, we will always assume that D includes all the constants occurring in S and $\bar{V}$.

Proposition 3.3.9 (Domain-Independence) *Let n be largest number of variables and constants occurring in any of the conjunctive queries $S, V_1, \ldots, V_k$. If there exists a domain[2] D_0 s.t. $|D_0| \geq n(n+1)$, and $crit_{D_0}(S) \cap crit_{D_0}(\bar{V}) = \emptyset$, then for any domain D, s.t. $|D| \geq n(n+1)$, $crit_D(S) \cap crit_D(\bar{V}) = \emptyset$.*

We now discuss how to decide query-view security for conjunctive queries S and $\bar{V}$. Our goal is to check dictionary-independent security, hence we need to check whether $crit_D(S) \cap crit_D(\bar{V}) = \emptyset$, and we assume that the domain D is "large enough". The previous proposition gives us an exponential time algorithm: pick a domain D_0 with $n(n+1)$ constants, then enumerate exhaustively all instances $I \subseteq D_0$ and tuples $t \in I$, checking whether t is a critical tuple for S, and for $\bar{V}$. This also shows that the query-view security problem is in complexity class Π_2^p. (Recall that NP is the class of problems that can be expressed as $\{z \mid \exists y\, \phi(y, z)\}$ where the "certificate" y has length polynomial in z and ϕ is PTIME computable. Complexity class Π_2^p consists of problems that can be expressed as $\{z \mid \forall x \exists y\, \phi(x, y, z)\}$ where x, y are polynomial in z and ϕ is PTIME computable.)

Theorem 3.3.10 *The problem of deciding, for conjunctive query Q, whether a tuple $t \notin crit(Q)$ is Π_2^p-hard (query complexity).*

This is a non-trivial result, whose proof uses a lengthy reduction from the $\forall\exists$3-CNF problem, and is omitted. We illustrate with an example why computing $crit(Q)$ is non-obvious. Clearly, any critical tuple t must be an homomorphic image of some subgoal of Q.

[2]For conjunctive queries without order predicates it suffices to pick the domains D_0, D with size $\geq n$. When order predicates are allowed, then we need n fresh constants between any two constants mentioned in the queries, which leads to $n(n+1)$.

But the following example shows the converse is not true:

$$Q() :- R(x, y, z, z, u), R(x, x, x, y, y)$$

Consider the tuple $t = R(a, a, b, b, c)$, which is a homomorphic image of the first subgoal. Yet t is not critical. Indeed, let I be any database s.t. $Q(I) = true$. Then the first subgoal must be mapped to t. But that means that both x and y are mapped to a. Thus the second subgoal must be mapped to the tuple $t' = R(a, a, a, a, a)$ and then $t' \in I$. Then the first subgoal can also be mapped to t', hence t is not critical.

Next, we show that deciding whether $crit(S) \cap crit(V) = \emptyset$ is at least as hard as deciding whether a tuple t is not critical for a query Q. Indeed, if we define $V = Q$, and $S() :- t$ (i.e. S simply checks for the presence of the tuple t), then $t \notin crit(Q)$ iff $crit(S) \cap crit(V) = \emptyset$. For the former we have claimed that it is Π_2^p-hard. In summary:

Theorem 3.3.11 *The problem of deciding whether a conjunctive query S is secure w.r.t. to a set of conjunctive views $V_1, \ldots V_k$ is Π_2^p-complete (query complexity).*

A practical algorithm For practical purposes, one can check $crit(S) \cap crit(\bar{V}) = \emptyset$ and hence $S \mid \bar{V}$ quite efficiently. Simply compare all pairs of subgoals from S and from $\bar{V}$. If any pair of subgoals unify, then $\neg S \mid \bar{V}$. While false positives are possible, they are rare: this simple algorithm would correctly classify all examples in this chapter.

3.3.4 Proof of the Fundamental Theorems

The technique used in the proof of Theorems 3.3.5 and 3.3.8 is of independent interest, and we present it here. Throughout this subsection we will fix the domain D and denote the set of tuples with tuples$(D) = \{t_1, \ldots, t_n\}$. Recall our notation from Sec. 3.2: $x_1 = \Pr[t_1], \ldots, x_n = \Pr[t_n]$. Hence, a probability distribution Pr is given by a set of numbers $\bar{x} \in [0, 1]^n$.

The Boolean Case, Single View We first prove both theorems for the case of boolean queries; moreover, we will consider a single view, rather than a set of views. Given a boolean query Q, we denote by $\Pr[Q]$ the probability that Q is true on a randomly chosen database

instance. Recall from Equations (3.1) and (3.2) that this probability is given by:

$$\begin{aligned} \Pr[Q] &= \sum_{\{I \in inst(D) | Q(I) = true\}} \Pr[I] \\ \Pr[I] &= \prod_{t_i \in I} x_i \cdot \prod_{t_j \notin I} (1 - x_j) \end{aligned} \tag{3.5}$$

Therefore $\Pr[Q]$ is given by a polynomial in the variables $x_1, \ldots, x_n$, which we denote $f_Q(x_1, \ldots, x_n)$ or $f_Q(\bar{x})$.

Example 3.3.12 Let $D = \{a, b\}$, and consider the boolean query:

$$Q() : -R(a, x), R(x, x)$$

In this case $\text{tuples}(D) = \{t_1, t_2, t_3, t_4\}$, where $t_1 = R(a, a)$, $t_2 = R(a, b)$, $t_3 = R(b, a)$, and $t_4 = R(b, b)$. Then Q can be written as the following DNF formula:

$$Q = t_1 \vee (t_2 \wedge t_4)$$

To compute f_Q one enumerates all 16 database instances $I \subseteq \text{tuples}(D)$. Q is true on 12 of them: $\{t_2, t_4\}$, $\{t_2, t_3, t_4\}$, ... For each of them we apply Eq.(3.5). This results in a sum of 12 expressions:

$$f_Q = (1 - x_1)x_2(1 - x_3)x_4 + (1 - x_1)x_2x_3x_4 + \ldots$$

After simplification we obtain: $f_Q = x_1 + x_2x_4 - x_1x_2x_4$. Let $Q' : -R(b, a)$ (so that $f_{Q'} = x_3$), and consider the boolean formula $Q \wedge Q'$. The polynomial $f_{Q \wedge Q'}$ is equal to $f_Q \times f_{Q'}$, i.e. $(x_1 + x_2x_4 - x_1x_2x_4)x_3$ because Q and Q' depend on disjoint sets of tuples.

Before we prove the two theorems we notice that query-view security for boolean queries can be restated as follows. Given boolean queries S and V, $S \mid_{\Pr} V$ iff:

$$f_{S \wedge V}(\bar{x}) = f_S(\bar{x}) \times f_V(\bar{x}) \tag{3.6}$$

where $\bar{x}$ corresponds to Pr. Indeed, this represents precisely Equation (3.4) for one specific choice of s and v, namely $s = true$ and $v = true$. One can show that if Eq.(3.4) holds for $(true, true)$, then it also holds for the other three combinations, $(false, true)$, $(true, false)$, $(false, false)$. Thus, $S \mid_{\Pr} V$ holds precisely if (3.6) holds.

We now restate the two Theorems for the boolean case:

Theorem 3.3.5 Let D be a domain, S a query, and V a view. Then:

$\forall \bar{x} \in [0,1]^n . f_{S \wedge V}(\bar{x}) = f_S(\bar{x}) \times f_V(\bar{x})$ iff $crit_D(S) \cap crit_D(V) = \emptyset$.

Theorem 3.3.8 Let D be a domain. If S and V are monotone boolean queries, then:

$\exists \bar{x} \in (0,1)^n . f_{S \wedge V}(\bar{x}) = f_S(\bar{x}) \times f_V(\bar{x})$ implies

$\forall \bar{x} \in [0,1]^n . f_{S \wedge V}(\bar{x}) = f_S(\bar{x}) \times f_V(\bar{x})$.

The crux of the proof relies on a close examination of the polynomials f_Q. The properties we need are summarized below. Their proofs are straightforward and are omitted:

Proposition 3.3.13 *Let $f_Q = Pr[Q]$, where Q is a boolean formula in $t_1, \ldots, t_n$. Then f_Q is a polynomial in the variables $x_1, \ldots, x_n$ with the following properties:*

1. *For each $i = 1, \ldots, n$, the degree of x_i is ≤ 1.*
2. *For each $i = 1, \ldots, n$, the degree of x_i is 1 iff $t_i \in crit_D(Q)$. (In Example 3.3.12, $crit_D(Q) = \{t_1, t_2, t_4\}$ and indeed x_1, x_2, x_4 have degree 1, while x_3 has degree 0.)*
3. *If $crit_D(Q_1) \cap crit_D(Q_2) = \emptyset$ then $f_{Q_1 \wedge Q_2} = f_{Q_1} \times f_{Q_2}$.*
4. *Choose values in $[0,1]^{n-1}$ for all variables except for one, x_i: f_Q becomes a polynomial of degree ≤ 1 in x_i. Then, if Q is a monotone boolean formula, the coefficient of x_i is ≥ 0. In Example 3.3.12, the coefficient of x_4 in f_Q is $x_2 - x_1 x_2$, which is always ≥ 0 when $x_1, x_2 \in [0,1]^2$.*
5. *Let Q_0 be the boolean formula obtained from Q by setting $t_n = false$, and Q_1 be the boolean formula obtained by setting $t_n = true$. Then $f_{Q_0} = f_Q[x_n = 0]$ and $f_{Q_1} = f_Q[x_n = 1]$. In example 3.3.12, $Q_0 = t_1$ and $f_Q[x_4 = 0] = x_1$; similarly $Q_1 = t_1 \vee t_2$ and $f_Q[x_4 = 1] = x_1 + x_2 - x_1 x_2$.*

We prove now Theorem 3.3.5 for the boolean case.

Proof: Assume first that $crit_D(S) \cap crit_D(V) = \emptyset$. Then $f_{S \wedge V} = f_S \times f_V$, by Proposition 3.3.13, item 3. Assume now that $\forall \bar{x} \in [0,1]^n . f_{S \wedge V}(\bar{x}) = f_S(\bar{x}) \times f_V(\bar{x})$ holds. Then the polynomials $f_{S \wedge V}$ and $f_S \times f_V$ must be identical. In particular, f_S and f_V cannot have a common variable x_i, otherwise its degree would be 2. Hence $crit_D(S)$ and $crit_D(V)$ cannot have a common tuple (by Prop. 3.3.13 item 2). □

Next we prove Theorem 3.3.8 for the boolean case.

Proof: Consider the polynomial $g_{S,V} = f_{S\wedge V} - f_S \times f_V$. We show by induction on the number n of tuples in tuples(D) that $\forall \bar{x} \in [0,1]^n$, $g_{S,V}(\bar{x}) \geq 0$. It holds trivially for $n = 0$. For $n > 0$, $g_{S,V}$ is a polynomial of degree ≤ 2 in x_n, and the coefficient of x_n^2 is negative: this follows from Proposition 3.3.13 item 4 and the fact that S, V are monotone. For $x_n = 0$, the polynomial in $n-1$ variables $g_{S,V}[x_n = 0]$ corresponds to the boolean formulas $S[t_n = false]$, $V[t_n = false]$ (item 5 of the proposition), hence we can apply the induction hypothesis and obtain that $g_{S,V} \geq 0$ for $x_n = 0$. Similarly, $g_{S,V} \geq 0$ for $x_n = 1$, since now it corresponds to the boolean formulas $S[t_n = true]$, $V[t_n = true]$. Furthermore, since $g_{S,V}$ has degree ≤ 2 and the coefficient of x_n^2 is ≤ 0, it follows that $g_{S,V} \geq 0$ for every $x_n \in [0,1]$. This completes the inductive proof. Now assume that for some $\bar{x} \in (0,1)^n$, $g_{S,V}(\bar{x}) = 0$. We will prove that $crit_D(S) \cap crit_D(V) = \emptyset$. Assume by contradiction that $t_i \in crit_D(S) \cap crit_D(V)$ for some tuple t_i. Then $g_{S,V}$ is a polynomial of degree 2 in x_i, with a negative coefficient for x_i^2, which has at least one root in $(0,1)$. It follows that $g_{S,V}$ must be < 0 either in $x_i = 0$, or in $x_i = 1$, contradicting the fact that $g_{S,V} \geq 0$ for all $\bar{x} \in [0,1]^n$. □

The Boolean Case, Multiple Views Let $\bar{V} = V_1, \ldots, V_n$ be n boolean views. The next step is to show that $S \mid_{\Pr} \bar{V}$ forall Pr iff $S \mid_{\Pr} V_i$ forall $i = 1, \ldots, n$ and all Pr. For simplicity we prove this for $n = 2$.

Proof: For the 'only if' direction we prove Eq.(3.4) directly. To show $\Pr[S(I)=s \wedge V_2(I)=v_2] = \Pr[S(I)=s] \times \Pr[V_2(I)=v_2]$ we notice:

$$\begin{aligned} \Pr[S(I)=s \wedge V_2(I)=v_2] &= \sum_{v_1} \Pr[S(I)=s \wedge V_1(I)=v_1 \wedge V_2(I)=v_2] \\ \Pr[V_2(I)=v_2] &= \sum_{v_1} \Pr[V_1(I)=v_1 \wedge V_2(I)=v_2] \end{aligned}$$

then we use the fact that $S \mid_{\Pr} (V_1, V_2)$ to complete the argument. For the 'if' direction, we need to check $\Pr[S(I) = s \wedge V_1(I) = v_1 \wedge V_2(I) = v_2] = \Pr[S(I) = s] \times \Pr[V_1(I) = v_1 \wedge V_2(I) = v_2]$. Using Theorem 3.3.5 for the boolean, single view case, it suffices to check $crit_D(S) \cap crit_D(V) = \emptyset$ where $V(I)$ is the boolean query $V_1(I) = v_1 \wedge V_2(I) = v_2$. This follows from $crit_D(V) \subseteq crit_D(V_1) \cup crit_D(V_2)$, and the assumption, $crit_D(S) \cap crit_D(V_i) = \emptyset$ for $i = 1, 2$. □

The Non-boolean Case We now generalize to non-boolean queries. Given a k-ary query Q, let $t_1, \ldots, t_m$ be all k-tuples over the domain D ($m = |D|^k$). For each $i = 1, \ldots, m$, define Q_i^b the boolean query $Q_i^b(I) = (t_i \in Q(I))$; that is, it checks whether t_i is in Q. Notice that $crit_D(Q) = \bigcup_i crit_D(Q_i^b)$, and if Q is monotone then Q_i^b is monotone for $i = 1, \ldots, m$.

Given a domain D and probability distribution, the following is easy to check, by applying directly the Definition 3.3.1. For any query S and views $\bar{V} = V_1, \ldots\ V_k$:

$$S \mid_{\Pr} \bar{V} \text{ iff } \forall i, j, l. S_i^b \mid_{\Pr} V_{j,l}^b$$

Here $V_{j,l}^b$ denotes $(V_j)_l^b$. This immediately reduces both theorems to the boolean case.

3.4 Modeling Prior Knowledge

So far we have assumed that the adversary has no knowledge about the data other than the domain D and the probability distribution Pr provided by the dictionary. Next we consider security in the presence of prior knowledge, which we denote with K. Our standard for security compares the adversary's knowledge about the secret query S before and after publishing the views $\bar{V}$, but always assuming he knows K. In the most general case K is any boolean statement on the database instance I. For example it can be a key or foreign-key constraint, some previously published views, or some general knowledge about the domain. K is thus any boolean predicate on the instance I, and we write $K(I)$ whenever I satisfies K. To avoid introducing new terminology, we will continue to call K a *boolean query*. We do not however restrict K by requiring that it be expressed in a particular query language.

3.4.1 Definition and Main Theorem

As before we assume domain D to be fixed. K is a boolean query, while S and $\bar{V}$ are arbitrary queries.

Definition 3.4.1 (Prior Knowledge Security) *Let Pr be a probability distribution on the tuples. We say that S is secure w.r.t. $\bar{V}$ under prior knowledge K if for every $s, \bar{v}$:*

$$Pr[S(I) = s \mid K(I)] = Pr[S(I) = s \mid \bar{V}(I) = \bar{v} \wedge K(I)]$$

We denote prior knowledge security by $K : S \mid_{Pr} \bar{V}$.

Applying Bayes' theorem reduces the above to:

$$\Pr[S(I) = s \wedge \bar{V}(I) = \bar{v} \wedge K(I)] \times \Pr[K(I)] = \Pr[S(I) = s \wedge K(I)] \times \Pr[\bar{V}(I) = \bar{v} \wedge K(I)] \quad (3.7)$$

Both the **prior knowledge** and the **relative security** applications mentioned in Sec. 3.1 are modeled as a security problem with prior knowledge. In the case of relative security, we take K to be the knowledge that the prior view has some given answer.

Theorem 3.3.5, which showed query-view security is equivalent to disjointness of the critical tuples, can be generalized for security with prior knowledge. We state the theorem for the boolean case, and will discuss specific generalizations to non-boolean queries and views.

Theorem 3.4.2 *Let D be a domain, $T = tuples(D)$, and K, S, V be arbitrary boolean queries. Then $K : S \mid_{Pr} V$ for all probability distributions Pr iff the following holds:*

COND-K *There exists sets of tuples T_1, T_2 and boolean queries K_1, K_2, V_1, S_2 s.t.:*

$$\begin{array}{rcl} T_1 \cap T_2 & = & \emptyset \\ K & = & K_1 \wedge K_2 \\ S \wedge K & = & K_1 \wedge S_2 \\ V \wedge K & = & V_1 \wedge K_2 \\ crit_D(K_1) \subseteq T_1 & & crit_D(K_2) \subseteq T_2 \\ crit_D(V_1) \subseteq T_1 & & crit_D(S_2) \subseteq T_2 \end{array}$$

Informally, the theorem says that the space of tuples can be partitioned into T_1 and T_2 such that property K is the conjunction of two independent properties, K_1 over T_1 and K_2 over T_2. In addition, assuming K holds, S just says something about the tuples in T_2 (and nothing more about T_1). Similarly, when K holds, V just says something about T_1 (and nothing more about T_2).

By itself, this theorem does not result in a practical decision procedure, because it is too general. We show, however, how it can be applied to specific applications, and in particular derive decision procedures.

3.4.2 Applying Prior Knowledge

Application 1: No prior knowledge As a baseline check, let's see what happens if there is no prior knowledge. Then $K = true$ and condition **COND-K** says that there are

two disjoint sets of tuples T_1 and T_2 such that $crit_D(S) \subseteq T_2$ and $crit_D(V) \subseteq T_1$. This is equivalent to saying $crit_D(S) \cap crit_D(V) = \emptyset$, thus we recover Theorem 3.3.5 for boolean queries.

Application 2: Keys and foreign keys The notion of query-view secrecy is affected by keys and foreign-keys constraints K. For an illustration, consider the boolean query: $S() : -R(a, b)$, and the boolean view $V() : -R(a, c)$. Here a, b, c are distinct constants. We have $S \mid_{\Pr} V$ for any Pr, because $crit_D(S) = \{R(a, b)\}$ and $crit_D(V) = \{R(a, c)\}$ are disjoint. But now suppose that the first attribute of R is a key. Then by knowing V we know immediately that S is false, which is a total information disclosure, hence $K : S \mid V$ does not hold.

We apply now Theorem 3.4.2 to derive a general criterion for query-view secrecy in the presence of key constraints K. Given a domain D, define the following equivalence relation on tuples(D): $t \equiv_K t'$ if t and t' are tuples over the same relation, and they have the same key. In the example above, we have $R(a, b) \equiv_K R(a, c)$, and $R(a, b) \not\equiv_K R(d, b)$ for a new constant d. Given a query Q, denote $crit_D(Q, K)$ the set of tuples t s.t. there exists a database instance I that satisfies the key constraints K and $Q(I) \neq Q(I - \{t\})$. The following criterion can be proven from Theorem 3.4.2 and shows how to check $K : S \mid \bar{V}$.

Corollary 3.4.3 *Let K be a set of key constraints, D a domain, and S, $\bar{V}$ be any queries. Then $S \mid_{Pr} \bar{V}$ for any Pr iff $\forall t \in crit_D(S, K)$, $\forall t' \in crit_D(\bar{V}, K)$, $t \not\equiv_K t'$. In particular, the problem whether $K : S \mid_{Pr} V$ for all Pr is decidable, and Π_2^p-complete.*

As a simple illustration, in the previous example, we have $crit_D(S, K) = \{R(a, b)\}$, $crit_D(V, K) = \{R(a, c)\}$, and $R(a, b) \equiv_K R(a, c)$, hence it is not the case that $K : S \mid_{\Pr} V$ for all Pr. Foreign keys can be handled similarly, however the corresponding decidability and complexity result holds only when the foreign keys introduce no cycles.

Proof: (Sketch) We give the proof for the boolean case only: the generalization to non-boolean queries is straightforward. The proof relies on two observations. Let $U_1, U_2, \ldots$ be the $\equiv_K$ equivalence classes on tuples(D). For each U_i, denote L_i the predicate saying "at most one tuple from V_i can be in the instance I". For example if $U_i = \{t_1, t_2, t_3\}$ set

$$L_i = (\neg t_1 \wedge \neg t_2 \wedge \neg t_3) \vee (t_1 \wedge \neg t_2 \wedge \neg t_3) \vee (\neg t_1 \wedge t_2 \wedge \neg t_3) \vee (\neg t_1 \wedge \neg t_2 \wedge t_3)$$

Then K is $L_1 \wedge L_2 \wedge \ldots$ The first observation is that whenever we split K into $K_1 \wedge K_2$ as in **COND-K** each of the two subexpressions must be a conjunction of some L_i's. It follows that every equivalence class U_i intersects either $crit_D(K_1)$ or $crit_D(K_2)$ but not both. The second observation is that there exists S_2 s.t. $crit_D(S_2) \subseteq T_2$ $S \wedge K = K_1 \wedge S_2$ iff $crit_D(S, K) \subseteq T_2$. Indeed, in one direction, we notice first that if I satisfies K, then $S(I) = K_1(I \cap T_1) \wedge S_2(I \cap T_2) = S_2(I \cap T_2)$. Then for any $t \in crit_D(S, K)$, there exists I satisfying K s.t. $S(I) \neq S(I - \{t\})$, hence $S_2(I \cap T_2) \neq S_2((I - \{t\}) \cap T_2)$, which implies $t \in T_2$. For the other direction, define $S_2(I) = K_2(I) \wedge S(I \cap T_2)$ (obviously $crit_D(S_2) \subseteq T_2$) and let's show that $S \wedge K = K_1 \wedge S_2$. Let I be an instance s.t. $(S \wedge K)(I)$ is true; in particular I satisfies K, hence $crit_D(S, K) \subseteq T_2$ which means $S(I) = S(I \cap T_2)$. The claim follows from the fact that $K_2(I)$ is true. With these two observations the proof of the corollary is straightforward and omitted. The decidability and complexity is shown with an argument similar to that used in Theorem 3.3.11. □

Application 3: Cardinality Constraint. What happens if the adversary has some partial knowledge about the cardinality of the secret database? This is quite common in practice. For example the number of patients in a hospital is likely to be between 100 or 1,000, but not 2 and not 1,000,000. In this case K is a cardinality constraint, such as "there are exactly n tuples in I" or "there are at most n tuples" or "at least n tuples". Surprisingly, there are no secure queries when the prior knowledge involves any cardinality constraints! This follows from Theorem 3.4.2 since K cannot be expressed as $K_1 \wedge K_2$ over disjoint sets of tuples, by a simple counting argument, except for the trivial case when $T_1 = \emptyset$ or $T_2 = \emptyset$. Hence, no query is perfectly secret w.r.t. to any view in this case, except if one of them (S or V) is trivially true or false.

Application 4: Protecting Secrets with Knowledge Sometimes prior knowledge can protect secrets! Take any queries S, $\bar{V}$, and assume that S is not secure w.r.t. $\bar{V}$. Suppose now that we disclose publicly the status of every tuple in $crit_D(S) \cap crit_D(\bar{V})$. That is, for each common critical tuple t we announce whether $t \in I$ or $t \notin I$. If we denote with K this knowledge about all common critical tuples, then Theorem 3.4.2 implies that $K : S \mid_{\Pr} \bar{V}$ for any Pr, as we show below. For a simple illustration, assume $S() : -R(a, -)$

and $V() :- R(-, b)$. They are not secure because $crit_D(S) \cap crit_D(V) = \{R(a, b)\}$. But now suppose we disclose that the pair (a, b) is not in the database, $R(a, b) \notin I$, and call this knowledge K. Then $K : S \mid_{\mathrm{Pr}} V$. The same is true if we publicly announce that $R(a, b)$ *is* in the database instance. We prove this formally next:

Corollary 3.4.4 *Let K be such that $\forall t \in crit_D(S) \cap crit_D(\bar{V})$, either $K \models t \in I$, or $K \models t \notin I$. Then, for every Pr, $K : S \mid_{Pr} \bar{V}$.*

Proof: We will prove this for two boolean queries S, V only: the general case follows easily. Let $T_1 = crit_D(S) \cap crit_D(V)$, and $T_2 = \text{tuples}(D) - T_1$. Let $K_1 = K$, $K_2 = true$, $S_2 = S$, $V_1 = V \wedge K$. Then the conditions of Theorem 3.4.2 are satisfied, hence $K : S \mid_{\mathrm{Pr}} V$ for any Pr. □

Application 5: Prior Views. Suppose Alice already published a view U (there may have been leakage about S, but she decided the risk was acceptable). Now she wants to publish another view V, and she wonders: will I leak any *more* information about S?

Using Theorem 3.4.2 we give below a decision procedure for the case of conjunctive queries, but only when U is a boolean query. This is a limitation, and due to the fact that both sides of the formula (3.7) are linear in S and $\bar{V}$, but not in K: this made it possible to generalize statements from boolean queries S, V to arbitrary ones, but not for K. To simplify the statement, we also restrict S and V to be boolean: these, however, can be generalized to arbitrary conjunctive queries.

Corollary 3.4.5 *Let U, S, V be boolean conjunctive queries. Then $U : S \mid_{Pr} V$ for every probability distribution Pr iff each of the queries can be split into the following:*

$$\begin{aligned} U &= U_1 \wedge U_2 \\ S &= S_1 \wedge S_2 \\ V &= V_1 \wedge V_2 \end{aligned}$$

such that the sets $crit_D(U_1) \cup crit_D(S_1) \cup crit_D(V_1)$ and $crit_D(V_2) \cup crit_D(S_2) \cup crit_D(V_2)$ are disjoint, and $U_1 \Rightarrow S_1$ and $U_2 \Rightarrow V_2$. Hence, $U : S \mid_{Pr} V$ is decidable.

The proof follows rather directly from Theorem 3.4.2 and is omitted. For a simple illustration consider:

$$U :- R_1(a, b, -, -), R_2(d, e, -, -)$$

$$S \ :- \ R_1(a,-,-,-), R_2(d,e,f,-)$$
$$V \ :- \ R_1(a,b,c,-), R_2(d,-,-,-)$$

Here S is not secure w.r.t. either U or V. However, $U : S \mid V$. By giving out U we already disclosed something about S, namely $R_1(a,-,-,-)$. By publishing V in addition we do not further disclose any information.

3.4.3 Proof of Theorem 3.4.2

Proof: (sketch) For boolean queries, $K : S \mid_{\Pr} V$ can be expressed as follows:

$$\Pr[S \wedge V \wedge K] \times \Pr[K] = \Pr[S \wedge K] \times \Pr[V \wedge K]$$

Using the notation f_Q for a boolean query Q (see Sec. 3.3.4), this becomes:

$$f_{S\wedge V\wedge K}(\bar{x}) \times f_K(\bar{x}) = f_{S\wedge K}(\bar{x}) \times f_{V\wedge K}(\bar{x}) \tag{3.8}$$

We need to prove that (3.8) holds for any $\bar{x} \in [0,1]^n$ iff **COND-K** holds. For that we need the properties of f_Q in Proposition 3.3.13 plus three more. Call any multi-variable polynomial $g(\bar{x})$ of degree ≤ 1 in each variable a *boolean* polynomial if $\forall \bar{x} \in \{0,1\}^n$, $g(\bar{x})$ is either 0 or 1. Clearly, any polynomial f_Q is a boolean polynomial.

Proposition 3.4.6

1. *If g is a boolean polynomial then there exists a unique boolean formula Q s.t. $g = f_Q$.*
2. *Let Q be a boolean formula, and suppose f_Q is the product of two polynomials $f_Q = g \times h$. Then there exists a constant $c \neq 0$ s.t. both cg and $\frac{1}{c}h$ are boolean polynomials.*
3. *If $f_Q = f_{Q_1} \times f_{Q_2}$ then $crit_D(Q_1) \cap crit_D(Q_2) = \emptyset$.*

We can now prove the equivalence of (3.8) to **COND-K**. Assume (3.8) holds for every $\bar{x} \in [0,1]^n$, i.e., this is an identity of polynomials. Then f_K divides $f_{S\wedge K} \times f_{V\wedge K}$. Hence $f_K = g \times h$ where g divides $f_{S\wedge K}$ and h divides $f_{V\wedge K}$. By Prop. 3.4.6 we can assume that g, h are boolean, hence $f_K = f_{K_1} \times f_{K_2}$ for some boolean formulas K_1, K_2, and moreover we have $K = K_1 \wedge K_2$ and $crit_D(K_1) \cap crit_D(K_2) = \emptyset$. Since f_{K_1} divides $f_{S\wedge K}$, we can write the latter as $f_{S\wedge K} = f_{K_1} \times f_{S_2}$, for some boolean query S_2, which implies $S \wedge K = K_1 \wedge S_2$. Similarly,

f_{K_2} divides $f_{V \wedge K}$, hence we can write the latter as $f_{V \wedge K} = f_{V_1} \times f_{K_2}$ for some query V_1. Finally, substituting in (3.8) and simplifying with $f_{K_1} \times f_{K_2}$ we get $f_{S \wedge V \wedge K} = f_{V_1} \times f_{S_2}$. It follows that f_{V_1} and f_{S_2} have no common variables, hence $crit_D(V_1) \cap crit_D(S_2) = \emptyset$. Define $T_1 = crit_D(K_1) \cup crit_D(V_1)$ and $T_2 = \text{tuples}(D) - T_1$. Then it follows that $crit_D(K_2) \subseteq T_2$ and $crit_D(S_2) \subseteq T_2$, completing the proof of **COND-K**.

For the other direction, assume **COND-K** is satisfied and let's prove (3.8). We have:

$$\begin{aligned} f_{S \wedge V \wedge K} &= f_{(K_1 \wedge V_1) \wedge (K_2 \wedge S_2)} = f_{K_1 \wedge V_1} \times f_{K_2 \wedge S_2} \\ f_K &= f_{K_1} \times f_{K_2} \\ f_{S \wedge K} &= f_{K_1} \times f_{S_2 \wedge K_2} \\ f_{V \wedge K} &= f_{V_1 \wedge K_1} \times f_{K_2} \end{aligned}$$

and (3.8) follows immediately. □

3.5 Relaxing the definition of security

Our standard for query-view security is very strong. It classifies as insecure query-view pairs that are considered secure in practice. In many applications we can tolerate deviations from this strong standard, as long as the deviations are not too large. We discuss briefly two directions for a more practical definition of security. The first strategy is a numerical measure of information disclosure, and the second, based on [36], uses a substantially different assumption of database probabilities which effectively ignores certain minute disclosures.

3.5.1 Subsequent work on practical query-view security

Following the original publication of this work [104], the author, along with Nilesh Dalvi and Dan Suciu, analyzed query-view security under a substantially different probabilistic model which can permit a relaxed notion of security termed *practical* security. For comparison purposes, we provide here a brief overview of the setting and main results for this approach, referring the reader to [36] for a full treatment of the topic.

To capture practical query-view security we adopt a new probability distribution over databases. In this model, individual tuples have a uniform probability of occurring in the database, but the probability of each tuple t is now such that the expected size of the relation

instance R is a given constant S (different constants may be used for different relation names). As the domain size n grows to ∞, the expected database size remains constant. Hence, in the case of directed graphs (i.e., a single binary relation R), the probability that two given nodes are connected by an edge is S/n^2. Denoting by $\mu_n[Q]$ the probability that a boolean query Q is true on a domain of size n, our goal is to compute $\mu_n[Q \mid V]$ as $n \to \infty$.

We propose as a definition of practical security $\lim_n \mu_n[Q \mid V] = 0$. This is justified as follows. The adversary faces a large domain. For example, if he is trying to guess whether *"John Smith"* is an employee, then he has only a tiny probability of success: $1/n$ where n is the size of the domain. On the other hand, the size of the database is much smaller, and the adversary often knows a good approximation. This definition relaxes the previous definition of security for sensitive queries Q.

In [36] we show that $\lim_n \mu_n[Q \mid V]$ for *conjunctive queries* Q and V always exists and provide an algorithm for computing it. The key technical lemma is to show that, for each conjunctive query Q there exists two numbers c, d s.t. $\mu_n[Q] = c/n^d + O(1/n^{d+1})$. Moreover, both d and c can be computed algorithmically. Since $\mu_n[Q \mid V] = \mu_n[QV]/\mu_n[V]$, the main result follows easily.

With this condition of practical security in mind we distinguish the following cases:

Perfect query-view security This is the condition analyzed in this paper. $V \mid Q$ can be rewritten as $\mu_n[Q \mid V] = \mu_n[Q]$ for all n large enough. Here V provides no information about Q.

Practical query-view security $\lim_{n\to\infty} \mu_n[Q \mid V] = 0$. This implies that the difference of probabilities is zero in the limit (since $\lim_n \mu_n[Q] = 0$ for all practical purposes). For finite n, V may in fact contain some information for answering Q, but it is considered negligible in this model.

Practical Disclosure $0 < \lim_{n\to\infty} \mu_n[Q \mid V] < 1$. Disclosure is non-negligible in this case. Our main result allows us to compute this quantity in terms of expected database size S.

3.6 Encrypted Views

Encryption is increasingly being used to protect both published data and data stored in the DBMS. In many scenarios [102, 5, 78, 4], data is encrypted at the attribute level, and analyzing the disclosure of these somewhat unusual "views" is an important open research question. (See Section 2.5 for examples.)

Encrypted views can be modeled in our framework. One way is to model an encrypted view of a relation as the result of applying a perfect one-way function f to each attribute. Because our goal is to study the logical security of a view, we assume idealized properties of the encryption primitive: namely that given $f(x)$ it is impossible to recover x, and that f is collision free. Under these assumptions an encrypted view is essentially an isomorphic copy of the original relation. Clearly, such a view provides information that can be used to answer queries. For example, $Q_1() : -R(x, y), R(y, z), x \neq z$ is answerable using such a view. Query $Q_2() : -R(a, x)$ is not answerable using the view, but substantial information is nevertheless leaked. It can be shown that no secret query S is secure w.r.t. an encrypted view V.

3.7 Related Work

3.7.1 Database security

Chapter 2 thoroughly addressed the relationship between conventional database access control, statistical databases, and disclosure.

The authors of [55] study formal definitions of privacy in the context of privacy preserving data mining. In this setting the goal is to permit accurate data mining models to be built over aggregates while preventing disclosure of individual items. Here the published view is the result of applying a randomization operator to data values or a distribution of data values. It is shown that a known information-theoretic definition of privacy may permit certain disclosures, and they propose an extended measure to account for this drawback.

A protection mechanism for relational databases was proposed in [10] based on a probabilistic definition of security similar to our own. An algorithm for deciding query-view security was not known, and relative security and quantified security were not addressed.

In [11] the same authors focus on comparing probabilistic independence (closely related to our security criterion) with algebraic independence, but are not concerned with a decision procedure for probabilistic independence.

3.7.2 Alternative models

Query-view security could be defined by comparing the entropy [123] of S with the conditional entropy of S given V. If this is done for all possible answers $s, \bar{v}$ (computing the entropy of event $S = s$, and $S = s$ given $\bar{V} = \bar{v}$) then the condition is equivalent to our security criterion. However, it would be more common to compare the entropies of the events defined by S and V, aggregating over the answers to S and V. This would result in a criterion strictly weaker than ours (see Example 2.5.6).

The goal of the probabilistic relational model [71, 84] is to model statistical patterns in huge amounts of data. The issues addressed are learning models from existing data, modeling statistics about a given database (e.g. to be used by a query optimizer), and inferring missing attribute values. These techniques do not provide us with means to reason about information disclosure, which is independent of a particular data instance.

3.7.3 Theoretical connections

A query is independent of an update if the application of the update cannot change the result of the query, for any state of the database. Detecting update independence is useful for maintenance of materialized views [17, 53, 90] and efficient constraint checking [76]. Deciding whether a tuple t is critical for a query Q (a notion we defined and studied in Section 3.3.3) is equivalent to deciding whether Q is independent of the update that deletes t from the database. Update independence is undecidable for queries and updates expressed as datalog programs [90], but has been shown decidable for certain restricted classes of queries like conjunctive queries with comparisons [17, 53, 90]. The tight bounds shown in this paper for deciding *crit*(Q) constitute an interesting special case for update independence.

The so-called FKG inequality [67] is a theoretical result about the correlation between events in a probability space. It is closely related to our security criterion, and can be used

to show that $\Pr(V \wedge S) \geq \Pr(V)\Pr(S)$, for monotone boolean properties V and S. However, it says nothing about when equality holds, and its inductive proof offers little insight. Our Theorem 3.3.8 reproves this inequality and furthermore proves the necessary and sufficient condition for equality to hold.

Another topic that connects logic to probability theory are the 0-1 laws [60, 61], which hold for a logical language if, for each formula, the probability that a formula is true converges to either 0 or 1 as the size of the domain tends to infinity. Our definition of query-view security is not related to 0-1 laws: our domain size does not grow to infinity but remains fixed and we are concerned about the effect of one formula (the view) on another (the secret query).

3.7.4 Subsequent work

Following the original publication of this work [104], the author, along with Nilesh Dalvi and Dan Suciu, has considered a relaxed notion of query-view security based on a substantially different probability distribution over databases [36]. The approach was described briefly in Section 3.5.1, although the main results are not included in this dissertation.

In [43] it is assumed that a set of views has been published to the adversary, regardless of disclosure about a secret query. A new view, considered for publication, is then evaluated for additional disclosure. (We consider such a scenario in Section 3.4.2) The authors study a version of query-view security similar to our own, but also consider weaker variants. They provide complexity bounds for these decision problems under general probability distributions, and for more expressive integrity constraints.

Chapter 4

CONFIDENTIALITY IN DATA EXCHANGE

The techniques described in the previous chapter assist the data owner in deciding which views of the database can be safely shared to avoid partial disclosures and inferences. The association of subjects with their authorized views of the database constitutes an access control policy. To enforce such a policy–that is, to ensure that subjects receive authorized views, and no other portions of the database–the owner must employ a protection mechanism. The present chapter describes a distributed protection mechanism which depends on encryption.

4.1 Introduction and Overview

Recall from Section 1 that distributed data exchange is characterized by multiple data owners sharing data with potentially large numbers of intermediaries and final recipients. For the following discussion, we focus on a single data owner seeking to share a database D with a large number of subjects. The database is initially stored using a trusted system administered by the owner. The subjects, denoted $s_1 \ldots s_n$, have varying access rights to the database, where subject s_i is authorized to see view v_i on D. The subjects also have systems capable of storing and processing data, but these systems are not controlled by the owner, and therefore not trusted. Although subject s_i is authorized to see $v_i(D)$, the subject may actually be interested in computing the answer to a different query, q_i. We assume that q_i is answerable using $v_i(D)$.

4.1.1 Protection alternatives

We distinguish between four alternative architectures a data owner can employ to share data and enforce access control policies.

Client-server In a client-server architecture, the data owner stores the database on a

trusted server which responds to requests by subjects for access to data. This is typical of modern database systems as well as file systems that enforce access control.

- A subject s_i, after authenticating, posts a request for access to the database, in the form of their query q_i.
- For each such request, the owner determines whether the query can be answered, given the subject's access rights.
- If admissible, the server executes the query and returns the results to the subject.

Gifford described this as an *active protection mechanism* [72] because an active process is placed between a client and protected data. Query execution is performed at the server, and only query results are transmitted to the subject. Note that it is never the case that a data item a subject is not authorized to see is sent to a subject.

View materialization and publishing This architecture is similar to that above, but authorized views are computed and transmitted to subjects in anticipation of actual requests. Interaction between owner and subjects consists of the following:

- The data owner computes the safe view $v_i(D)$ for each subject s_i.
- The data owner publishes $v_i(D)$, sending it to the subject over a secure channel.
- The subject stores $v_i(D)$ data locally, and then computes q_i using its own query execution resources.

Again, it is never the case that a data item the subject is not authorized to see is sent to a subject.

Controlled publication Controlled publication is the focus of this chapter. Gifford described this as a *passive protection mechanism* [72] because there is no system restricting the subjects' access to data.

- Using access control policies for each subject, and techniques described below, the data owner computes a single, partially-encrypted database.

- The data owner publishes this single document to all subjects, e.g. by posting on a web server.
- The data owner transmits key-sets to each subject, over a secure channel.
- Subject s_i retrieves the published data. Using their authorized keys, the subject can decrypt the data to derive the authorized view $v_i(D)$, or selectively decrypt in order to compute q_i.

Any data item a client is not authorized to see must be encrypted, since the protected database is released to all clients.

Trusted computing base This architecture assumes that a trusted operating environment executes at the client. This typically requires tamper-resistant hardware. A practical instance of this is a smart card, among others [6]. Effectively, the entire database D may be transmitted to each subject's secure environment, over a secure communication channel. The client posts queries to the trusted component, which is responsible for access decisions. This allows data items a client is not authorized to see to be stored at the client, since proper access will be negotiated by the trusted process running at the client. Recent research into secure operating environments and XML data protection is described as related work in Section 4.9.5. The feasibility of current designs based on this architecture depends heavily on enhanced hardware capabilities of the future.

4.1.2 Comparative analysis

Each of the described architectures may be desirable for certain applications. In particular, the choice of architecture will depend on properties of the database, the access policies to be enforced, the number of clients, the processing resources of owner and subject, and the available network bandwidth.

The focus of this chapter is controlled data publishing, which has a number of advantages for distributed data exchange. This architecture is a very efficient means for broadcasting data to many users, reducing the overall network bandwidth, and simplifying the publication

mechanism by allowing a single protected database to be published once, and used by all subjects. Storage and dissemination of data is separated from access control restrictions, and untrusted intermediaries can assist in storing and sharing data in encrypted form. This relieves the requirement that the owner's server be online and available for access control decisions, and it also relieves the processing burden on the server, moving load to the subject. Finally, this architecture supports query privacy because subjects can process their authorized data without revealing to the server what data is accessed, how it is processed, and when it is processed.

The architecture does have some limitations, however. It is impossible to prevent an authorized but malicious subject from decrypting the data it is authorized to see and releasing it to unauthorized recipients. This is a basic limitation that applies to each of the other architectures as well. In addition, revocation is difficult using controlled data publication. Once access is granted to a subject and data is published, that data cannot be retracted. If, at a later point in time, a subject's authorization changes, that change can only be reflected in the publication of subsequent versions of the database.

4.1.3 System Overview and Chapter Organization

The remainder of this chapter describes a framework for controlled data publication, supporting large numbers of subjects with varied access rights. The three main conceptual components of the publication framework (illustrated in Figure 4.1) are:

1. Policy Queries – We define a language for specifying access control policies, which can express high level notions like "A psychologist should have access to the records of patients she examines". The policy language defined here is based on query languages specifically designed for XML data. An example policy query is shown on the left of Figure 4.1. Policy queries are described by example in Section 4.2, and then formally in Section 4.6.

Policy queries are evaluated on a database, resulting in (i) cryptographic keys associated with subjects or groups of subjects, and (ii) an intermediate representation of a protected XML document, called a *tree protection*. The keys are transmitted to authorized subjects

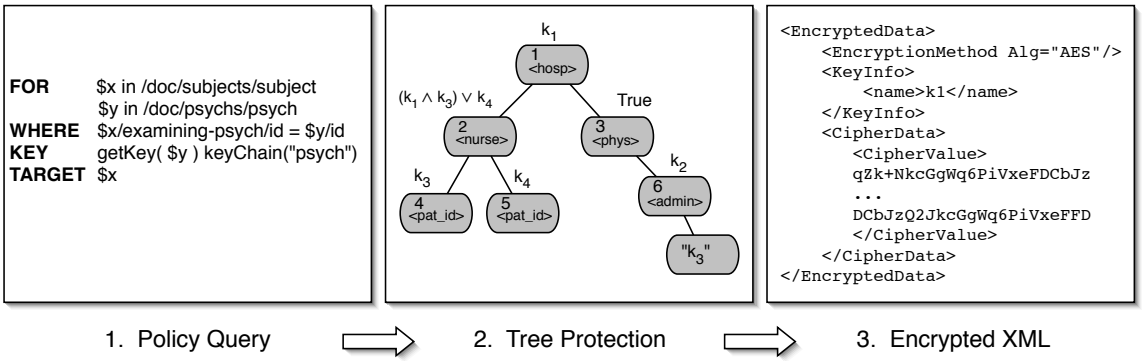

Figure 4.1: The protected data publishing framework.

and act as digital capabilities [41], permitting access to specific portions of the database. The tree protection relates keys and the accessibility to data that they allow.

2. Tree Protection – An example tree protection is shown in the middle of Figure 4.1. A tree protection is a logical model for protected data which annotates the XML document tree with "guard" formulas over symbolic key expressions. The model is powerful enough to express complex policies, has a precise semantics, and admits some basic logical optimizations which later result in significant space savings for the published data. Tree protections are described in Section 4.3.

3. Encrypted Data – Finally, the tree protection is translated into a partially-encrypted XML document. The encrypted instance can be represented using the recent W3C Recommendation *XML Encryption Syntax* [51] as a physical format. Sample encrypted output is pictured on the right of Figure 4.1. The present work adapts and extends known techniques including secret sharing [119, 12], bounded key encryption [63, 64], and random sequences [125]. Generation of the final encrypted XML is described in Section 4.4 and the security of encrypted instances is discussed in Section 4.5.

At this point in the framework, the partially-encrypted XML document may be published freely. It contains the entire accessible database and enforces *all* access policy queries. Keys

are transmitted to subjects, who download the data and process it. A subject can decrypt the entire authorized view, or can decrypt the data selectively, using a query language, and supplying appropriate keys. Techniques for processing encrypted data are discussed in Section 4.7.

In Section 4.8 the performance of the framework is evaluated. We study the size of the partially encrypted documents, and the encryption/decryption speed. We show that these measures are reasonable, and can be dramatically improved using a combination of logical and physical optimizations. The chapter concludes with a thorough discussion of related research areas in Section 4.9.

4.2 Policy query examples

This section describes a motivating scenario illustrating the language for writing policy queries. We defer a complete description of the language syntax and semantics to Section 4.6. Recall that policy queries are evaluated by the database owner on an XML data instance, in a secure environment, before publishing the access-controlled version of the database, and will be re-evaluated or updated when the database changes, to produce a new version of the published data.

4.2.1 Examples

We now motivate our techniques by presenting a scenario of controlled data publishing. What follows is inspired from actual challenges faced by biologists at the University of Washington in meeting their goals of data dissemination while satisfying trust and security constraints. The example includes a number of participants, the trust and privacy issues between them, and a series of example policy queries that exercise the capabilities of our framework.

In this scenario a group of *primary researchers* enlist the support of *technicians* in carrying out medical and psychological tests on willing experimental *subjects*. Once the data is analyzed, the primary researchers submit their results to the conference *publisher*. In addition, experimental data must be published so that *competing researchers* can use it. Finally, an *auditor* checks if certain privacy regulations are enforced.

The policy queries use an extension of XQuery with a KEY and a TARGET clause. These clauses are used to associate symbolic keys with regions in the XML data that those keys will protect. The first policy query is motivated by the relationship between primary and competing researcher:

```
Policy Query 4.2.1
  SUFFICIENT
  FOR       $x in /doc/subjects/subject
  KEY       getKey("registration"),
            $x/analysis/DNAsignature/text()
  TARGET    $x/analysis
```

This query declares that users with the two keys in the KEY clause will be granted access to the analysis target. The first key is an *exchange* key, named "registration": the getKey() construction retrieves the key named "registration", or, if one doesn't exist, generates a new secure key and stores it for future use. The second key is taken from the data itself: namely the user must know the value of the DNAsignature field in order to access the entire analysis element. Notice that this query fires for all subject elements. The "registration" key will be the same for each target node, while the DNAsignature value will likely be different for each target node.

The intent of Policy Query 4.2.1 is to allow competing researchers who have registered (and thus acquired the registration key) to access the analysis of all subjects with a DNAsignature they can provide. This severely impedes the competing researchers from doing uncontrolled scans of all DNA samples, but allows them to verify data for the DNA samples that they already have.

```
Policy Query 4.2.2
  SUFFICIENT
  FOR       $x in /doc/subjects/subject
  KEY       getKey( $x ) keyChain("imageKeys")
  TARGET    $x/analysis/brain-scan
```

This query generates a new key for each subject, and grants access to brain-scan data to users in possession of that key. The argument to getKey is now a node, $x. In addition a keyChain is specified: this is a convenient way to organize keys. The key for $x is retrieved from–or stored into–the key chain named "imageKeys". When the data owner decides to

grant access to the brain-scan of a specific subject to some user, she looks up the key associated to that subject in the keychain "imageKeys" and sends it to the user through a secure channel . She can thus have very fine-grained control over how users access the data.

Notice that the targets of Policy Queries 4.2.1 and 4.2.2 overlap. The SUFFICIENT keyword means that satisfaction of *any* rule grants access to the common target. In particular, brain-scan data can be accessed either with the keys specified in Query 4.2.1 or with the key in Query 4.2.2.

Policy Query 4.2.3

```
SUFFICIENT
FOR       $x in /doc/subjects/subject
          $y in /doc/psychs/psych
WHERE     $x/examining-psych/id = $y/id
KEY       getKey( $y ) keyChain("psych")
TARGET    $x
```

This query simply says that a psychologist examiner is allowed to see all subjects he examined. A new key is generated for each psychologist (or retrieved, if it already exists), in the keychain named "psych", and that key grants access to all subjects examined by that psychologist. Notice that if a subject was examined by multiple psychologists[1] then each will have access to that subject: this query results in self-overlap, in addition to the overlap with previous queries.

Next we show a more intricate policy query, motivated by the legal requirement of protecting personal identity data such as name and social security number. Lab technicians need access to some of the subjects' data, e.g. age, sex, etc., but not to the identity data. However, subjects with blood type "AB-" are very rare: only one or two are encountered each year. A technician could trace the identity of such a subject from the exam-date/year information. The two policy queries below grant technicians conditional access to various data components:

Policy Query 4.2.4

---

[1]This happens when subject has more than one examining-psych subelements.

```
SUFFICIENT
FOR       $x in /doc/subjects/subject
WHERE     $x/blood-type != "AB-"
KEY       getKey( "tech1") keyChain("technicians")
TARGET    $x/age, $x/sex,
          $x/blood-type, $x/exam-date/year

SUFFICIENT
FOR       $x in /doc/subjects/subject
KEY       getKey("tech1") keyChain("technicians")
WHERE     $x/blood-type = "AB-"
TARGET    $x/sex, $x/blood-type
```

The first policy query says that the key "tech1" grants access to four fields (age, sex, blood-type, and exam-date/year), but only of subjects with blood type other than "AB-". For the latter, the second query grants access only to sex and blood-type.

Finally, an auditor wants to verify that HIV tests are protected. Under the lab's policy, only registered users have access to the HIV test, hence the auditor's query is:

Policy Query 4.2.5

```
NECESSARY
FOR       $x in /doc/subjects/subject
KEY       getKey("registration")
TARGET    $x/analysis/tests/HIV
```

Notice that this query starts with the NECESSARY keyword: it means that *only* users having the key named "registration" (same as in Query 4.2.1) have access to the analysis/tests/HIV data. We provide a complete description of the policy language syntax and semantics in Section 4.6.

4.3 The Tree Protection

This subsection describes the logical model for protecting an XML tree, which plays a central role in the protected publishing framework: it is the output of policy queries, the input to the physical encryption procedure, and the data model for the client's queries. A tree protection consists of a tree where nodes are "guarded" by keys or sets of keys. Such a protected tree limits access in the following way: only users with an admissible set of keys can access an element. Without a qualifying set of keys, the element's name, its attributes, and its children are all hidden. We present this formally next.

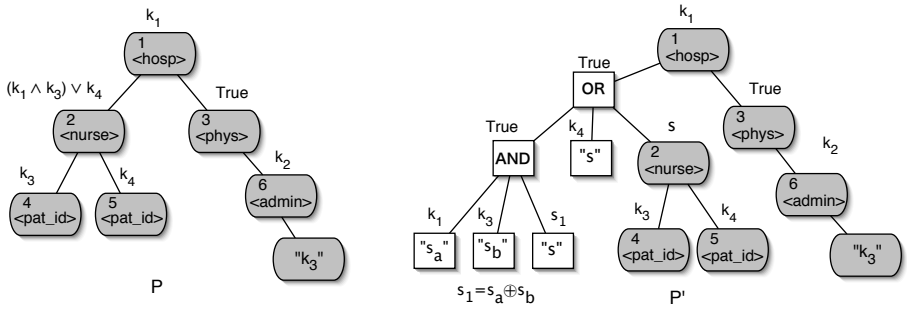

Figure 4.2: A tree protection P (Example 4.3.1), and an equivalent normalized protection P' (Example 4.3.4). The white nodes are metadata nodes.

XML Trees We model an XML document as a node-labeled tree t with internal nodes labeled from an alphabet of element and attribute names, and leaves labeled from a set of values. We denote the set of nodes with $\textsc{nodes}(t)$ and the set of edges with $\textsc{edges}(t)$, and $\textsc{value}(i)$ the value of a leaf node i. Given two nodes $i, j \in t$, we write $i \prec j$ when i is a proper ancestor of j and $i \preceq j$ for the ancestor-or-self relation.

Keys We consider three kinds of keys in our model: exchange keys, inner keys, and data value keys. To simplify our discussion we fix the length of key values at 128 bits, but varying bit lengths are supported.

Exchange keys are stored by the data owner and communicated through secure channels to various clients. They have a public name, for identification, unrelated to their key value which is used for encryption and decryption. Referring to Sec. 4.2.1, examples of exchange keys are: "registration" in Query 4.2.1, and the subject's keys in Query 4.2.2, with system generated names like "subject030223". We denote by NamedKey the (finite) domain of exchange keys.

Inner keys are random numbers generated by the system during the encryption process, and stored in the XML data itself (as base64-encoded text): users can only learn them by accessing the XML data where they are stored. There are no inner keys illustrated in our motivating example because they are generated automatically by the system.[2] We denote

[2]Inner keys are needed to support complex access control policies, and are used in the normalization process described later in Sec. 4.3.2.

by LocalKey the (finite) domain of inner keys.

Data Value keys are all the text values, numbers, dates, etc., that can normally occur in an XML document. We derive a 224-bit string by using the UTF-8 encoding, padding it appropriately if necessary, and then applying a cryptographic hash function (e.g. SHA-224 [58]). This convention is publicly known and can be repeated by the user. We use data values as keys because, in some applications, access may be granted to users who know certain fields of a protected piece of data. For example, a user who knows a patient's name, address, and social security number may have access to the entire patient's record. We denote by DataValue the set of values. In our motivating example, the DNAsignature in Query 4.2.1 is a data value key. We write Key = NamedKey ∪ LocalKey ∪ DataValue.

XML values In our model the leaves of an XML document may carry either a *data value*, or an *inner key*. While the latter is encoded as a base64 string (thus we could model it as a data value), we distinguish it from an access control point of view. Users may acquire, by some independent means, certain data values with meaningful semantics: names, addresses, social security numbers, bank account numbers. But they have no way of learning an inner key, except by accessing an XML node where that inner key is stored as a value. Thus, in our model, the value of a leaf node $i \in \text{nodes}(t)$ is $\text{value}(i) \in \text{DataValue} \cup \text{LocalKey}$.

Metadata XML nodes We introduce an extension to the basic XML model, by allowing some additional metadata nodes in the tree. Their role is to express certain protections or to hold inner keys, and they are introduced automatically by the system. Formally, a *metadata XML tree* t_m over a tree t is obtained from t by inserting some *meta-data nodes*, which can be either internal nodes (element or attribute nodes), or leaf nodes, and may be inserted anywhere in the tree. Thus, $\text{nodes}(t_m)$ consists of meta-data nodes, plus $\text{nodes}(t)$: we call the latter *data nodes*. We assume that the meta-data nodes can be distinguished in some way from the data nodes, for example by using an XML namespace. (In figures, metadata nodes are white while data nodes are gray.) An operation, $\text{trim}(t_m) = t$, recovers the XML tree from the metadata tree, by removing the metadata nodes and promoting their children. For any XML tree t, a trivial metadata tree is t itself, having no metadata node: in this case $\text{trim}(t) = t$.

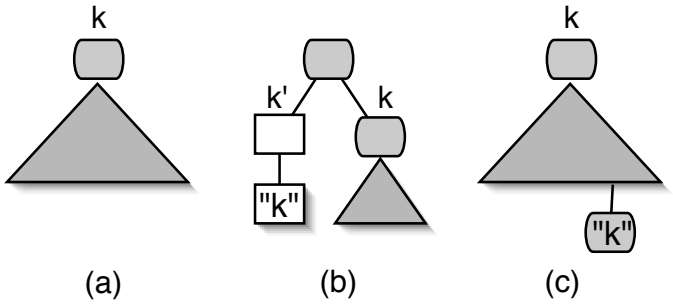

Figure 4.3: Typical usage patterns of tree protections, described in Example 4.3.2.

Tree Protection A *protection* over an XML tree t is $P = (t_m, \sigma)$ where t_m is a metadata tree over t and σ associates to each node $i \in \text{NODES}(t_m)$ a positive boolean *guard formula* σ_i over KEY, satisfying the following grammar (where $k \in \text{KEY}$):

$$\sigma := \quad \text{true} \mid \text{false} \mid k \mid \sigma \vee \sigma' \mid \sigma \wedge \sigma'$$

The intuition is that σ_i defines the key(s) that a user needs to present in order to gain access to the node i. But in order to reach the node i from the root, the user needs to also satisfy all formulas σ_j, for all $j \prec i$. This justifies the next definition: we call the *necessity formula*, φ_i of a node i to be $\varphi_i = \wedge_{j \preceq i} \sigma_j$.

Example 4.3.1 Figure 4.2 illustrates a protection $P = (t_m, \sigma)$. Each node $i \in t$ is annotated with its guard formula σ_i, using named exchange keys k_1, k_2, k_4, and data value key k_3. Guard formula σ_2 is equal to $(k_1 \wedge k_3) \vee k_4$ and necessity formula φ_6 is equal to $k_1 \wedge \text{true} \wedge k_2$. There are no metadata nodes in this case, so $t = t_m$.

Example 4.3.2 Figure 4.3 illustrates typical usages for the three kinds of keys. An exchange key is used as in (a): it simply protects a subtree. An inner key is shown in (b) where the white nodes are metadata nodes: the key k protects the right subtree, and the user can access it only by reading the left subtree, which in turn is locked with k'. A data value key is shown in (c): here k is a data value stored in the tree, for example a Social Security number, but the user must know it in order to access the tree.

4.3.1 Access Semantics

The access semantics of a tree protection P is defined formally by the *access function* $acc_P(K)$, which, given a set of keys K, returns a set of nodes accessible by a user who "knows" K.

Precisely what is revealed when a node is accessible deserves careful consideration, and Figure 4.4 provides illustration. When a node is inaccessible, its element name and all descendants are hidden, as in Figure 4.4(a). When a node is accessible, it means that the element name of the node is visible, and the existence and number of the node's child elements is apparent (although the children themselves may be protected). This is shown in Figure 4.4(b). We adopt this definition throughout, and in Section 4.4 we describe an encryption technique which enacts this notion of access.

This is only one possible definition of what access to a node entails. Other notions of access can easily be accommodated by the framework. For example, another possible convention would reveal the existence of child nodes, but not their number (Figure 4.4(c)) or may hide the existence of children all together (in Figure 4.4(d)). If an alternative access semantics is desired, it can be simulated by adopting the standard semantics and simply introducing additional metadata nodes. For example, to hide the number of children of an accessible node x, a metadata node m can be created and the tree protection modified so that m is the parent of the children of x, and m is the sole child of x. Access to the number of children is then controlled by permitting or denying access to m.

Now we describe the access function formally. The function will return only the *data* nodes that the subject can access: during the process described below she may also access metadata nodes, but we only care about which data nodes she can access. The input K will be restricted to $K \subseteq \textsc{NamedKey} \cup \textsc{DataValue}$, because before accessing the XML document a user can only know exchange keys and data values, not inner keys.

The function $acc_P(K)$ is described by an iterative process. The user currently has access to a set of nodes N (initially empty) and to a set of keys M (initially K). N may contain both data nodes and metadata nodes, while M may contain all types of keys, including inner keys. At each step she increases M by adding all values on the leaf nodes in N, and

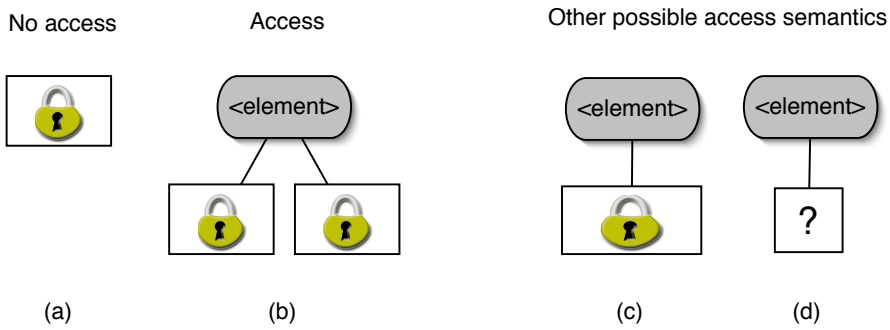

Figure 4.4: Various access semantics for a tree protection node.

increases N by adding all nodes that she can unlock by "using" the current keys in M. In order to unlock a node, she can either use the keys in M directly, or combine some keys in order to generate new keys. For example secret sharing protocols require the computation of a bit-wise XOR between two random keys, $r_1 \oplus r_2$. For our semantics we assume to be given a function $M' = \mathit{combine}(M)$ which, given a finite set of keys $M \subseteq \textsc{Key}$, returns a set of keys $M' \supseteq M$ which includes all allowed combinations of the random keys in M. The exact definition of *combine* may depend on the protocols: for our purpose, we will define it to be $\mathit{combine}(M) = M \cup \{r \oplus r' \mid r, r' \in M \cap \textsc{LocalKey}\}$. Other choices are possible, but one has to restrict *combine* to be computationally bounded, otherwise it may return the set of all random keys[3]. Finally, we need the following notation: for a set of keys M and positive formula φ over M we say $M \models \varphi$, if φ is *true* under the truth assignment derived from M by using keys in M as *true* propositional variables and assuming all others *false*. For example: $\{k_1, k_2, k_3\} \models k_4 \vee (k_1 \wedge k_2)$ but $\{k_1, k_2\} \not\models k_2 \wedge k_3$.

We can now define the function $\mathit{acc}_P(K)$ formally: $\mathit{acc}_P(K) = N \cap \textsc{nodes}(t)$, where $N \subseteq \textsc{nodes}(t_m)$ and $M \subseteq \textsc{Key}$ are the least fixed point of the following two mutually recursive definitions:

$$\begin{aligned} N &= \{i \mid i \in \textsc{nodes}(t_m), \mathit{combine}(M) \models \varphi_i\} \\ M &= K \cup \{\textsc{value}(i) \mid i \in N\} \end{aligned}$$

[3]The set of random keys is finite, e.g. 128-bit keys.

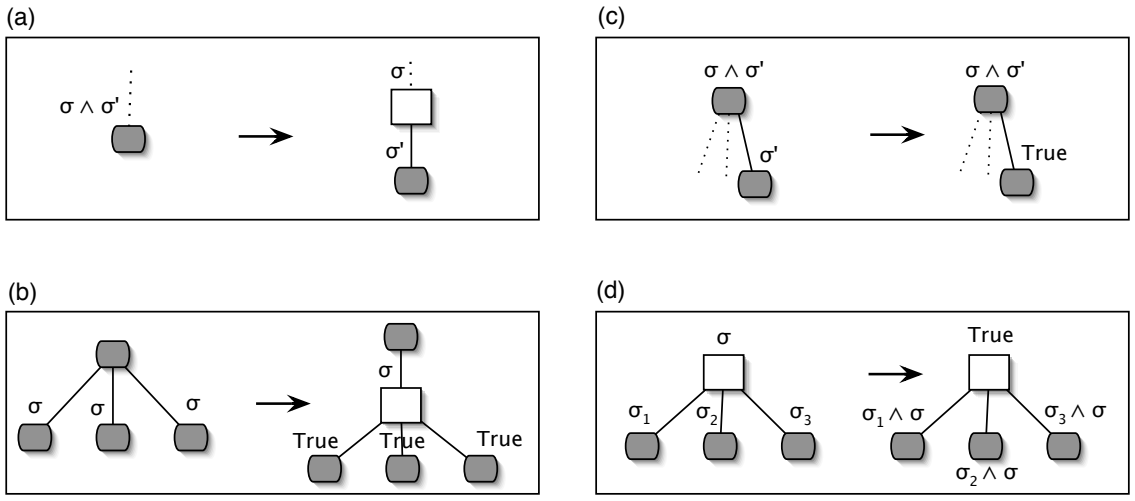

Figure 4.5: Protection rewritings for logical optimization: (a) formula split; (b) formula pull-up; (c) formula simplification; (d) formula push-down.

Finding this fixed point can be done with a standard iterative procedure[4] which corresponds to the informal description above.

Example 4.3.3 For the tree protection P illustrated in Figure 4.2, the following are values of $acc_P(K)$ for selected subsets $K \subseteq$ NamedKey. (The subtree of t returned by the access function is represented as a set of node identifiers.) $acc_P(\{k_1\}) = \{1,3\}$, $acc_P(\{k_2\}) = \{\}$, $acc_P(\{k_1,k_2\}) = \{1,2,3,4,6\}$, $acc_P(\{k_1,k_4\}) = \{1,2,3,5\}$, $acc_P(\{k_1,k_3\}) = \{1,2,3,4\}$, $acc_P(\{k_1,k_3,k_4\}) = \{1,2,3,4,5\}$.

Having defined semantics we can now define equivalence between two protections P, P' of the same XML tree t. Namely P and P' are equivalent (in notation, $P \equiv P'$) if for every set $K \subseteq$ NamedKey $\cup$ DataValue, $acc_P(K) = acc_{P'}(K)$. Notice that the two protections may use different metadata nodes: what is important is that the user can learn the same set of nodes from both protections, with any set of keys K.

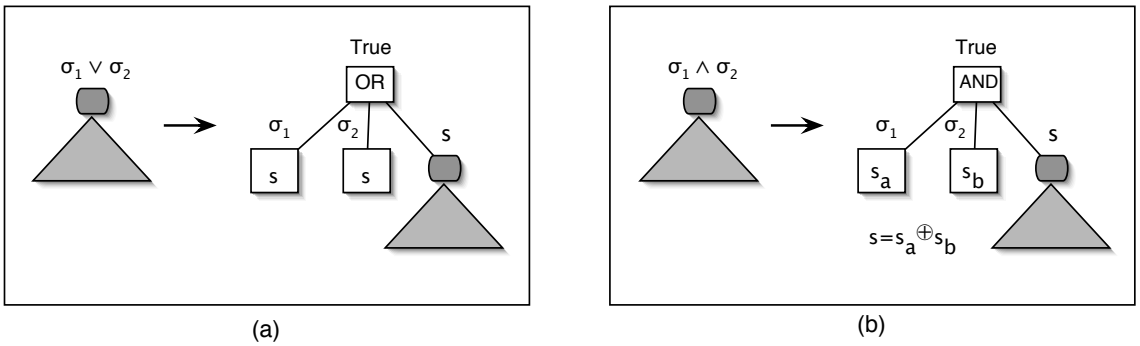

Figure 4.6: Rewriting formula conjunction and disjunction during tree protection normalization.

4.3.2 Rewrite Rules

We describe a set of local rewrite rules to be used in optimizations and normalization. It is easy to check that all rewrite rules are sound, i.e. each replaces one protection with an equivalent one. This can be verified in each case by showing that the access functions for the left and right protection are equal.

The rewritings in Figure 4.5 are intended to express logical optimizations. (The list is not exhaustive.) For example, in the left protection of rule (b) the same formula appears on many nodes: the right protection introduces a new metadata node, and uses that formula only once. In (c) the need for nested protections is eliminated: nested protections lead to nested encryptions, which increase the size of the output XML document significantly. Hence (c) can be viewed as a space-reducing optimization.

Figure 4.6 shows more rewriting rules that we use to normalize the protection before encrypting it as shown in Section 4.4. A protection is *normalized* if every formula σ_i is atomic (i.e. consisting of a single key, *true*, or *false*). These rules therefore transform a protection with complex key formulas (disjunctions and conjunctions) into a protection having only atomic key formulas. In the process new meta data nodes and new randomly-generated inner keys are introduced. We describe the rewritings below:

---

[4]This definition can be expressed in datalog, for example.

(a) *Disjunction*: To eliminate $\sigma_1 \vee \sigma_2$ a new random key, s, is generated and stored twice: once guarded with σ_1 and once with σ_2. The actual data is now guarded with s.

(b) *Conjunction*: To eliminate $\sigma_1 \wedge \sigma_2$ two new random keys, s_a and s_b, are generated and guarded with σ_1 and σ_2 respectively. The actual data is guarded with $s = s_a \oplus s_b$. A user needs to acquire both s_a and s_b in order to unlock s: knowing only s_a, for example, reveals nothing about s. This is a standard secret sharing protocol in cryptography [12].

Recall that in the definition of the access function $acc_P(K)$ the set of keys K is required to consist only of exchange keys and data values. Had we allowed K to contain inner keys too, then these two rewrite rules would not be sound: having the inner key s a user can access the protected trees on the right of Fig. 4.6, but learns nothing on the left. Our definition of the semantics, $acc_P(K)$, is such that it allows the system to introduce its own inner keys, as in Fig. 4.6.

Example 4.3.4 Figure 4.2 contains the normalized tree protection P' resulting from P after an application of rule (a) followed by an application of rule (b). A client in possession of key k_4 can access the `nurse` element by discovering key s in the metadata node. Alternatively, a client with both keys k_1 and k_3 can discover s_a and s_b and use them to compute s, thereby also gaining access to the `nurse` element.

4.4 Generating Encrypted XML

Given an XML document t and protection P, we now describe how to generate an encrypted XML document t' that *implements* P such that a user (or adversary) knowing a set of keys K will have efficient access to those nodes of t in $acc_P(K)$, and only those nodes. The first step is to apply logical rewritings to optimize the tree protection. The next step is to normalize P and obtain an equivalent protection P' for a metadata tree t_m. Once normalized, every node in the tree protection will be guarded by an atomic formula (no disjunction or conjunction), and encryption can be performed bottom-up.

4.4.1 Encrypted data format

The recent W3C Recommendation on XML Encryption Syntax and Processing [51] provides a standardized schema for representing encrypted data in XML form, along with conventions for representing cryptographic keys and specifying encryption algorithms.

The basic object is an XML element `EncryptedData` containing four relevant subelements: `EncryptionMethod` describes the algorithm and parameters used for encryption/decryption; `KeyInfo` describes the key used for encryption/decryption (but does not contain its value); `CipherData` contains the output of the encryption function, represented as base64-encoded text; `EncryptionProperties` contains optional user-defined descriptive data. The cipher text included in the `CipherData` element is the encryption of an XML element or element content. When the encrypted contents is itself an `EncryptedData` element, it is called nested encryption.

4.4.2 KeyInfo

In our framework, the content of the `KeyInfo` element contains its length in bits, and other fields that depend on the type of the key, as follows. For an *exchange key* it simply contains a `Name` subelement equal to its identifier. For an *inner key* it contains either one or two `Name` subelements: in the first case the `Name` is the local name of the inner key; in the second case the two inner keys with these names need to be XOR-ed. Recall from earlier in this section that for a *data value key*, its derived bitstring v is used for encryption and decryption. In this case, `KeyInfo` contains the following subelements: `path` denotes the path expression that leads to this data value; `concealment` contains $v \oplus r$ for a random bitstring r; `hash` contains $h(r)$, the result of the SHA-224 hash function [58] applied to r. It is computationally hard to reproduce v given the values of `concealment` and `hash`. However, a user with a proposed data value can easily check whether it matches the data value key. The user derives the bitstring w from their proposed value, computes $h(w \oplus (v \oplus r))$, and tests whether the result equals `hash`. This will be the case when $v = w$. This technique is more secure than an earlier encryption method described in [63, 64] for a similar purpose. In designing it, we were inspired by the techniques of [125] for search on encrypted data.

4.4.3 Nested encryption

Nested encryption of the protection P' is done by a straightforward recursive traversal of the metadata tree of P'. The encryption proceeds as follows. A node protected by the key *true* is simply copied to the output (after processing its children recursively). A node protected with the key *false* is removed from the output, together with all its children and descendants. A node protected with a key k is translated into an `EncryptedData` element with the following children: `EncryptionMethod` (in our case this is always AES with 128-bit keys), `KeyInfo`, which has the structure described above, and `CipherData`, which is the encryption of the node with the current key. This encryption process is consistent with the desired access semantics described previously: possession of a key guarding a node allows decryption to reveal the element name, and a collection of encrypted elements representing the children of the node.

Example 4.4.1 Figure 4.7 shows an XML instance constructed from the normalized protection P' of Figure 4.2. The `CipherValue` element contains bytes (encoded as base64 text) which may be decrypted to reveal the root element of the original tree. Once decrypted, the element name (`<hosp>`), and its attributes, are revealed. Its content however is still partially encrypted: the first child of the `<hosp>` element is another `EncryptedData` element, while the second child, `<phys>`, is unencrypted since it is not protected in P.

Compression Nested encryption can result in a significant size increase. We deal with this at a logical level by applying the rewriting rules in Sec. 4.3.2, and at the physical encryption level we can apply compression before encryption. We discuss this further in the context of the performance analysis in Section 4.8.

4.5 Security Discussion

The analysis of the security of the encrypted instances produced in this framework introduces many of the problems of disclosure analysis reviewed in Section 2.5. The confidentiality of published data depends on the security of the cryptographic functions employed and the soundness of the protocol, but in addition there may be subtle disclosures such as

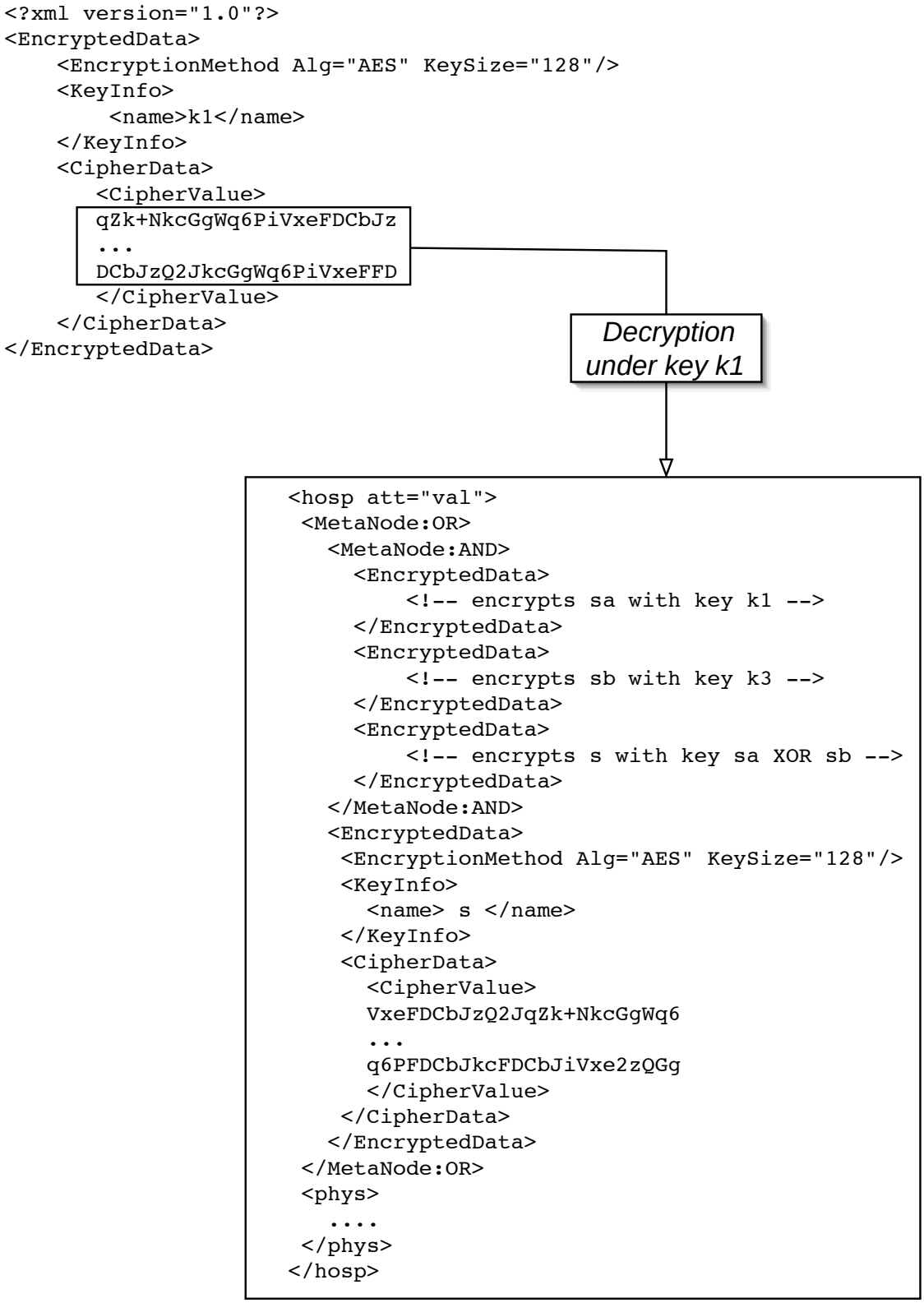

Figure 4.7: XML produced by applying nested encryption to the protection of Figure 4.2.

the size of the encrypted instances, the number of children of a node, and even clues about access policies.

One way to state a desired security property is as follows:

Property 4.5.1 (Security) *Suppose t is an XML document, P is a protection over t, and t' is the implementation of P over t described above. Then for any set of keys K:*

1. *if $x \in acc_P(K)$, there is an efficient algorithm for reproducing x from t' and K.*

2. *if $x \notin acc_P(K)$ then it is computationally infeasible to derive x from t' and K.*

The first statement is easy to verify: the access function, $acc_P(K)$ can be easily computed following its definition, for example by running a datalog program. The second statement is much more complex. It relies on the fact that each cryptographic construction we use is in itself secure. For example, in secret sharing a key s is computed as $s_a \oplus s_b$, and nothing at all can be deduced about s from either s_a or s_b in isolation. This protocol offers the strongest security guarantee (much stronger than any practical encryption algorithm) [12]. Our protocol for encrypting with data value keys is also secure. However, the security of the *combined* protocols requires a separate formal proof. This analysis is beyond the scope of this dissertation, but has been provided by others, as described below.

4.5.1 Provable protocol security

Abadi and Warinschi have recently performed a security analysis [2] of the core of the publishing framework described above, using techniques for the formal analysis of cryptographic protocols that the authors had developed in prior works [1, 99]. Their analysis applies to tree protections without data value keys, and with a more general use of secret sharing schemes (thresholds in place of simple disjunction and conjunction). Their work relates the abstract semantics of access (formally defined by the tree protection) to the information that may be derived by an adversary from the structured, partially-encrypted strings of bits that make up the published data. In particular, they prove that data that is protected according to the access function of a tree protection is indeed secret according to a strong computational notion of security.

4.5.2 Other disclosures

Not captured by Property 4.5.1 is the fact that an adversary may learn facts about the data without decrypting nodes. For example they will see the size of the encrypted ciphertext hiding a subtree. In some cases, they can count the number of encrypted children of a node, even if they cannot decrypt them. Furthermore, the metadata nodes present in the protected document may reveal aspects of the security policy, as they depend on the presence of disjunction and conjunction in the tree protections which in turn depend on the structure of policies. An accurate analysis of these subtle disclosures remains an open problem.

4.5.3 Data value key entropy

As described in Section 4.3, the bytes making up data value keys are derived in a deterministic way from values occurring in the database. The space of possible keys is therefore bounded by the domain of the values. For example, the first policy query in Section 4.2.1 used a data value key based on a patient's DNA signature. In many such cases the space of possible keys may be small, raising the risk of brute-force guessing attacks by an adversary. This is particularly dangerous since the database is published, and the adversary can attempt to guess the key repeatedly without detection. In the context of forensic DNA databases, the authors of [19] found that stored DNA entries often have less than 80-bits of entropy, making a brute force attack feasible. Other data values might have substantially smaller entropies. In general therefore, the use of data value keys alone to protect data is suspect. Instead, a data value key should be combined with conventional cryptographic keys (as it is in the example in Section 4.2.1). The conventional key can be given to a class of users who qualify for access to the data, but should be discouraged from doing uncontrolled scans of the the data. This creates a soft access restriction which we believe can be very effective in limiting data disclosure amongst authorized groups.

4.6 Policy queries: syntax and semantics

This section describes the syntax and semantics of the policy query language, which was illustrated by example in Section 4.2. The semantics has a number of subtleties: a *set* of

policy queries must evaluate to a *single* unified protection on the XML tree (as described in Sec. 4.3) and must therefore resolve possibly overlapping and contradictory policy queries.

4.6.1 Language Syntax

The general form of a policy query is:

```
[SUFFICIENT | NECESSARY]
FOR ... LET ... WHERE ...
KEY       keyExpr1, keyExpr2, ...
TARGET    targetExpr1, targetExpr2, ...
```

A policy query can be either a *sufficient* or a *necessary* query. The query contains an XQuery [18] FLWR expression (but without the RETURN clause) followed by a KEY and a TARGET clause. KEY expressions have the following form:

```
KEY   [path-expr] |
      [getKey(key-name) [keyChain(keychn-name)] ]
```

The first expression, path-expr, denotes a data value key, and must evaluate to a data value. The second expression, getKey(key-name), is an exchange key expression, optionally followed by a key chain name. If such a key exists in the keychain then it is retrieved; otherwise a new random 128-bit key is generated, and is associated with that name and keychain. The expressions targetExpr1, targetExpr2, ... are XPath expressions denoting nodes in the XML document.

4.6.2 Language Semantics

Intuitively, given an input XML document t, a policy query specifies a protection over t as follows. If the query is a *sufficient* query, then it says that a user holding the keys k1, k2, ... can unlock the target node. If the query is *necessary*, then it says that any user that can access the target must have the keys k1, k2, ... Typically, a data provider writes multiple protection queries, and evaluates all of them on the XML document t that it wants to publish, which results in a protection P for t that enforces *all* the queries. Such a protection may not exist. We say that the policy queries are *consistent for* t if a protection for t exists; we say that they are *consistent* if they are consistent for any t. Checking

consistency and constructing the protection P is non-obvious. We show how to do this next.

Policy queries → primitive rules The first step is to evaluate the policy queries on the XML document t and obtain a set of primitive rules. Given an XML document t, a *primitive sufficient rule* is a rule of the form $r_s = S \rightarrow e$, where S is a set of keys and $e \in \text{NODES}(t)$. Similarly, a *primitive necessary rule* is a rule of the form $r_n = e \rightarrow S$. Thus, a primitive rule applies to a particular tree t, and to a particular element of that tree. Given a tree t and a policy query, we evaluate the query to obtain a set of primitive rules on t, as follows. We first compute all variable bindings in the FOR...WHERE...LET... clauses: this computation is a standard step in any XQuery processor. For each such binding the key expressions in the KEY clause evaluate to some keys $k_1, k_2, \ldots$, and the target expressions in the TARGET clause evaluate to some nodes $v_1, v_2, \ldots$ For each descendant-or-self node e of some target node (i.e $v_i \preceq e$, for some $i = 1, 2, \ldots$) add the rule $\{k_1, k_2, \ldots\} \rightarrow e$, if the query was a sufficient query, or the rule $e \rightarrow \{k_1, k_2, \ldots\}$, if the query was a necessary query. Repeat this for each binding of the query, then for each policy query. The result is a set, R, of primitive rules for t.

Primitive rules → Protection We show here how to derive a protection P_R that "enforces" all primitive rules in a set of primitive rules R. The protection is over t itself, i.e. no metadata nodes are added (these are added later, during normalization). The intuition for the construction below is the following. The meaning of a sufficient rule $S \rightarrow e$ is that any user having the keys S can access the node e; a necessary rule $e \rightarrow S$ specifies that the user is not allowed access to e unless he has all keys in S. We seek a protection P_R that satisfies all rules in R, but it is easy to see that such a protection is not uniquely defined. For example if R contains only sufficient rules, then the True protection, where each guard formula is simply *true*, satisfies R: clearly this is not what we want from a set of primitive sufficient rules. Instead we define the meaning of R to be the *most restrictive* protection satisfying all rules. We make this formal next, using lattice-theoretic techniques [73].

Recall the definition of $acc_P(K)$ in Sec. 4.3.

Definition 4.6.1 (Primitive rule satisfaction) *Let P be a protection over metadata tree*

t.

- *For a sufficient primitive rule $r_s = S \rightarrow e$, P satisfies r_s (denoted $P \rhd r_s$) if $e \in acc_P(S)$.*
- *For a necessary primitive rule $r_n = e \rightarrow S$, P satisfies r_n (denoted $P \rhd r_n$) if for all K, if $e \in acc_P(K)$ then $S \subseteq K$.*

Define now $PS(R) = \{P \mid P = (t, \sigma), \forall r \in R, P \rhd r\}$. This is the set of all protections over t that satisfy all rules in R. Notice that we only consider protections over t, and do not allow additional metadata nodes. We define next the *most restrictive* protection in the set $PS(R)$ to be the greatest lower bound, for the following order relation. Recall that $\varphi_i = \bigwedge_{j \preceq i} \sigma_j$ is the necessity formula at node i (Sec. 4.3).

Definition 4.6.2 *Given two protections P and P' over the same metadata tree t, P is* more restrictive *than P', denoted $P \ll P'$, if for all nodes $i \in nodes(t)$, $\varphi_i \rightarrow \varphi'_i$ (the logical implication holds). The relation $\ll$ is a preorder*[5]*. For a set of protections S, $GLB(S)$ denotes the greatest lower bound under $\ll$.*

We can now define formally the meaning P_R of a set of primitive rules R to be $GLB(PS(R))$, when $PS(R) \neq \emptyset$, and to be undefined otherwise. In other words, the meaning of R is the most restrictive protection that satisfies all primitive rules in R. We show how to construct $GLB(PS(R))$, when $PS(R) \neq \emptyset$. In particular this construction proves that the greatest lower bound exists.

We partition the set of primitive rules into sufficient and necessary primitive rules: $R = R_s \cup R_n$. The following theorem summarizes the key properties that we need in order to compute the protection P_R. For a set of key expressions $S = \{\sigma_1, \ldots, \sigma_n\}$, the notation $\bigwedge S$ denotes $\sigma_1 \wedge \ldots \wedge \sigma_n$, and $\bigvee S$ denotes $\sigma_1 \vee \ldots \vee \sigma_n$.

Theorem 4.6.3 *Let t be a metadata tree, R_s be a set of primitive sufficient rules and R_n a set of primitive necessary rules on t. Then:*

- *If $GLB(PS(R_s \cup R_n))$ exists then it is equal to $GLB(PS(R_s))$.*
- *$GLB(PS(R_s))$ always exists, and is the protection defined as follows. For every node $i \in \text{NODES}(t)$, the key expression σ_i is given by: $\sigma_i = \bigvee\{\bigwedge S \mid \exists (S \rightarrow e) \in R_s, i \preceq e\}$ That is, the key formula for the node i is the disjunction of all key expressions S that are sufficient to unlock some descendant of i.*

[5]Reflexive and transitive.

- *$GLB(PS(R_s \cup R_n))$ exists iff the following* **Consistency Criterion** *is satisfied: For every pair of rules $(S \to e) \in R_s$ and $(e' \to S') \in R_n$, if $e' \preceq e$ (i.e. e' is a ancestor-or-self of e), then $S' \subseteq S$.*

Proof: Recall the preorder relationship $P \ll P'$ between two protections of the same tree (Definition 4.6.2). For any protection P let $[P)$ denote $\{P' \mid P \to P'\}$ and $(P]$ denote $\{P' \mid P' \to P\}$.

Recall that $PS(R)$ denotes the set of all protections P satisfying all rules in R. We examine next this set, as well as its greatest lower bound $GLB(PS(R))$, starting with a single sufficient rule, $r : S \to e$. In this case $P_r = GLB(PS(r))$ is the following. It has $\bigwedge S$ on the path from the root to e and $False$ everywhere else. It is obvious that P_r satisfies r, since with the set of keys S we can clearly access the node e in P_r. One can also see that, for any other protection P that satisfies r, we have $P_r \ll P$. Indeed, let i be a node s.t. $i \preceq e$. In P_r it is labeled with $\bigwedge S$; in P it is labeled with some formula σ_i, and recall that the necessity formula is $\varphi_i = \bigwedge_{j \preceq i} \sigma_j$. Since $e \in acc_P(S)$, we also have $i \in acc_P(S)$, which means that $S \models \varphi_i$ (since there are no metadata nodes in t, hence no keys to be learned besides S). This implies that $\bigwedge S \to \varphi_i$ is true. For other nodes i, which are not ancestors of e or e itself, the label in P_r is $false$, and $false \to \varphi_i$ also holds. Hence $P_r \ll P$.

More interestingly, we have in this case, $PS(r) = [P_r)$. Indeed, $PS(r) \subseteq [P_r)$ follows from the fact that $P_r = GLB(PS(r))$. For the other direction, we observe that any protection P s.t. $P_r \ll P$ satisfies the rule $S \to e$. Indeed, from $P_r \ll P$ we have that $\bigwedge S \to \varphi_i$ for every node $i \preceq e$, where φ_i is the necessity formula in P. Hence S unlocks the entire path from the root to e, and $e \in acc_P(S)$.

Set of sufficient rules We now examine the structure of $PS(R)$ and $P_R = GLB(PS(R))$ for a set of sufficient rules, $R = \{r_1, \ldots\ r_n\}$. We have:

$$\begin{aligned} PS(R) &= PS(r_1) \cap \ldots \cap PS(r_n) \\ &= [P_{r_1}) \cap \ldots \cap [P_{r_n}) \\ P_R &= GLB(PS(R)) \\ &= GLB([P_{r_1}) \cap \ldots \cap [P_{r_n})) \\ &= LUB(P_{r_1}, \ldots, P_{r_n}) \end{aligned}$$

The first line follows directly from the definition of $PS(R)$ (the set of protections that satisfy *all* rules in R). The last line is because $GLB([x) \cap [y)) = LUB(x, y)$ in any lattice. As before, $PS(R) = [P_R)$.

One necessary rule Consider now a single necessary rule in isolation, $r : e \rightarrow S$. The false protection, where each guard formula is simply $false$, denoted P_{bottom}, is in $PS(r)$, hence clearly the semantics here is $P_r = P_{bottom}$. Necessary rules in isolation are not interesting, since their most restrictive protection is to lock everything. For technical reasons denote with P'_r the protection having S on e and $True$ everywhere else.

Given a set of protections $\mathcal{P}$, we denote $LUB(\mathcal{P})$ the least upper bound, for the preorder $\ll$. For a single necessaryy rule r, it should be clear that $LUB(PS(r)) = (P'_r]$: this can be shown by reasoning similarly to the way we discussed a single sufficient rule. Moreover, for a set of necessary rules $R = \{r_1, \ldots\ r_m\}$ we have:

$$PS(R) = PS(r_1) \cap \ldots \cap PS(r_m) = (P'_{r_1}] \cap \ldots \cap (P'_{r_m}]$$

and, defining $P'_R = P'_{r_1} \wedge \ldots \wedge P'_{r_m}$, we have $PS(R) = (P'_R]$.

Mixed sufficient and necessary rules Now consider a set of sufficent and necessary rules, $R = R_s \cup R_n$. We describe its semantics, $P_R = GLB(PS(P_r))$. We know that $PS(R_s) = [P_{R_s})$ and $PS(R_n) = (P'_{R_n}]$. Then:

$$PS(R) = [P_{R_s}) \cap (P'_{R_n}]$$

There are two cases:

1) when $[P_{R_s}) \cap (P'_{R_n}] \neq \emptyset$. Equivalently: $P_{R_s} \subseteq P'_{R_n}$. In this case $GLB(PS(R)) = GLB([P_{R_s})) = P_{R_s}$.

2) when $[P_{R_s}) \cap (P'_{R_n}] = \emptyset$. In this case the set of rules is inconsistent, and its semantics is undefined.

Finally, we show how to check for Case 1 or Case 2. We need to check the following:

$$P_{r_1} \vee \ldots \vee P_{r_n} \subseteq P'_{r'_1} \wedge \ldots \wedge P'_{r'_m} \quad (*)$$

where $r_1, \ldots\ r_n$ are the sufficient rules and $r'_1, \ldots\ r'_m$ are the necessary rules. (*) is equivalent to:

$$P_{r_i} \subseteq P'_{r'_j} \text{ for every } i, j \quad (**)$$

To check (**) let $r_i : S \rightarrow e$ and $r'_j : e' \rightarrow S'$. There are two cases. First, if e' is an ancestor-or-self of e. Then (**) holds iff $S' \subseteq S$. Second, if e' is not an ancestor-or-self of e, then (**) always holds. This completes the proof. □

Evaluation Procedure This results in the following procedure for computing the protection P_R from a set of primitive rules R. First check the consistency criteria: if it fails, then P_R is undefined and the set of rules is inconsistent. Otherwise, we retain only the sufficient rules R_s, and construct the protection as follows. Given a node i, identify all rules $S_1 \rightarrow e_1$, $S_2 \rightarrow e_2, \ldots$ for which the target nodes $e_1, e_2, \ldots$ are either i or its descendants: then protect i with the key expression $(\wedge S_1) \vee (\wedge S_2) \vee \ldots$

Checking Consistency Statically

The procedure outlined above checks at runtime whether a set of queries is consistent for t. It is also possible to check at compile time whether a set of policy queries is consistent (i.e. for any input tree t). We show next how to reduce the problem to query containment for the XQuery language. Consider all pairs of sufficient and necessary policy queries Q_s and Q_n. We write them as:

$Q_s(S_1, ..., S_m, E)$:

```
FOR...LET...WHERE...
KEY S1, ..., Sm
TARGET E
```

$Q_n(S'_1, ..., S'_n, E')$

```
FOR...LET...WHERE...
KEY S1', ..., Sn'
TARGET E'
```

A set of policy queries is consistent for all XML documents if, for any pairs of queries Q_s, Q_n, and for every $i = 1, \ldots,$ (number of keys in Q_n), the query containment $Q \subseteq Q_1 \cup \ldots Q_m$ holds where Q, Q_i are defined in Figure 4.8.

$$
\begin{aligned}
Q(E, S_1, ..., Sm, Si') &:- Qs(E, S_1, ..., S_m), Qn(E', S'_1, ..., S'_i, ..., S'_n), E \preceq E' && (4.1)\\
Q_1(E, S_1, ..., S_m, S'_i) &:- Q(E, S_1, ..., S_m, S'_i), S_1 = S'_i && (4.2)\\
&... && (4.3)\\
Q_m(E, S_1, ..., S_m, S'_i) &:- Q(E, S_1, ..., S_m, S'_i), S_m = S'_i && (4.4)
\end{aligned}
$$

Figure 4.8: Queries for checking consistency statically.

Thus checking consistency is no harder than deciding containment of queries in the language considered. For the complete XQuery language the containment problem is undecidable (since it can express all of First Order Logic), and hence, so is the consistency problem. However, if one restricts the policy queries to a fragment of XQuery for which the containment problem is decidable, then the consistency problem is decidable. For example, [44] describes such a fragment for which the containment problem is Π_2^P-complete. Containment for XQuery was also studied in [50] with complexity results provided for a variety of sublanguages.

4.7 Data Processing

4.7.1 Querying Protected Data

A user holding a copy of the protected data instance P, and a set of keys K may access P naively by implementing the access function $acc_P(K)$ from Sec. 4.3. This, however, is hopelessly inefficient. We describe here a simple extension of XQuery that allows the user to access data selectively, and, moreover, guide the query processor on which keys to use where. The extension has a single construct: access(tag, k1, k2, ...) where tag is an XML tag and k1, k2, ... are key expressions of the following form:

getKey(key-name) | path-expr = value

The first denotes an exchange key, while the second a data value key.

Example 4.7.1 Consider Policy Query 4.2.1 from Sec. 4.2.1. Assume a physician downloaded the data, named it protectedData.xml, and needs to access the analysis element of a patient named "John Doe". Recall that this data is protected by both the "registration" key

and by the DNAsignature data value. The physician has the "registration" key, and can retrieve the DNAsignature from its local database, called patients.xml. She does this as follows:

```
FROM    $x in document("patients.xml")/
          patients/patient[name="John Doe"]
        $y in document("protectedData.xml")/
          subjects/subject/
          access(analysis, getKey("registration"),
          DNAsignature/txt()=$x/DNAsignature/txt() )
RETURN  $y
```

The query returns the analysis element.

This construct can be implemented in a query execution environment to decrypt only `EncryptedData` elements for which a qualifying set of keys is held, and then select those decrypted elements that match tag. Other optimizations that in addition avoid decrypting elements that do not match tag are also possible, but beyond the scope of the discussion.

4.8 Performance Analysis

Next we discuss the performance of a preliminary Java implementation. We begin with an input document t and protection P, generate the encrypted document t' enforcing P, and then process t' naively by reproducing t by decryption (assuming possession of all keys). We focus on the following metrics: time to generate t', the size of t' compared with t, and the time to reproduce t from t'.

Algorithm Choice and Experimental Setup We use a public Java implementation [23] of the Advanced Encryption Standard (AES) [34] with 128-bit keys. We tested other algorithms as well and while the ideal choice of algorithm is a complex issue[6] it is not a critical factor for the results below. We use three real datasets (Sigmod Record, an excerpt of SwissProt, and Mondial) for our experiments[7]. We consider basic protections which place single unique keys at all nodes on different levels of the document trees, and are named

[6]The choice of algorithm is a trade-off between raw encryption speed, key setup time, sensitivity of each of these to key length, in addition, of course, to the security of the algorithm.

[7]Available from the University of Washington XML Data Repository: `www.cs.washington.edu/xmldatasets`

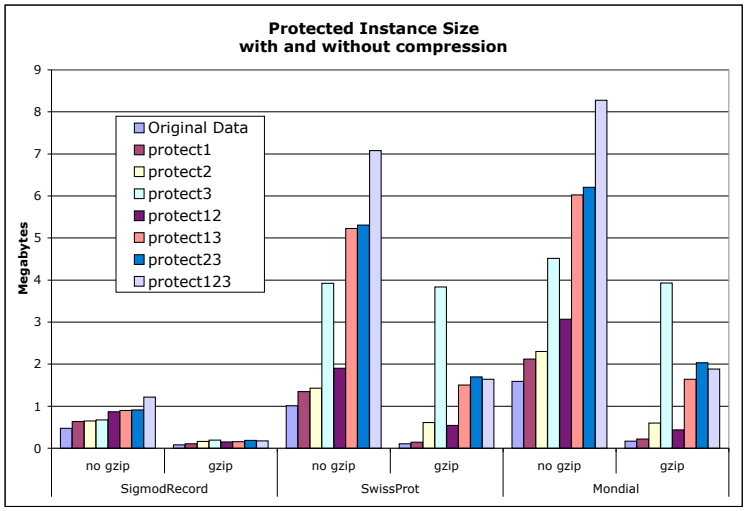

Figure 4.9: Size of protected documents, with and without compression.

accordingly. For example, P_1 guards the root with a single key, and P_{23}, guards nodes at level 2 and 3 with distinct keys (with $True$ everywhere else).

Protected document size The ciphertext output of a symmetric encryption algorithm is generally the same size as the cleartext input (modulo padding to block boundaries). However, in the case of encrypting XML, the cleartext uses a text encoding with 8 bits per character (UTF-8), while the ciphertext is binary data and is represented using 6 bits per character (base64 text), to conform with the Encryption standard [51]. This results in an immediate blow-up of 33% in the case of the simplest encrypted XML. The problem is compounded during nested-encryption, since the inflated representation of the ciphertext becomes the cleartext for another round of encryption. We address this difficulty by applying rewriting rules to avoid nested-encryption, and also by using compression.

Figure 4.9 presents the size of the encrypted instance t' for the three datasets and various protections, with and without compression. The protected instance can be considerably

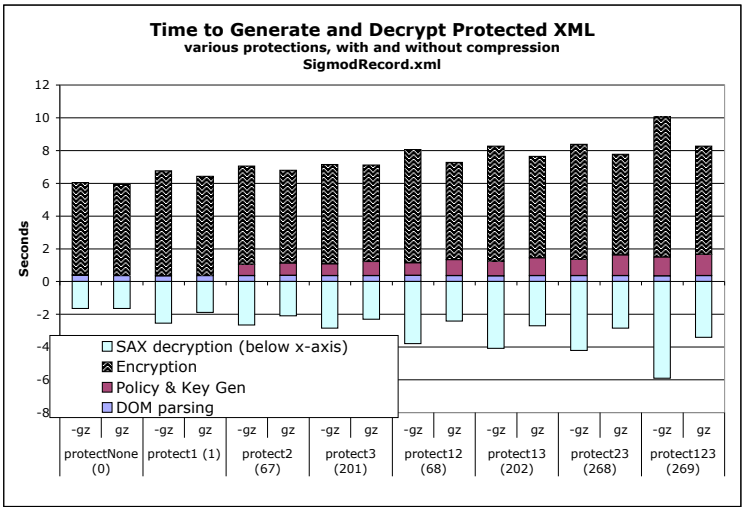

Figure 4.10: Processing time for generation and decryption of protected documents.

larger than the original, especially in the case of P_{123} which involves many keys and nested-encryption: 5 times the size of original for Mondial, 7 times for SwissProt for this protection. Applying rewriting rules (Sec. 4.3), when possible, can help however. Protection P_{13} can be seen as an approximation of the result of applying the rewriting rule in Figure 4.5(d) to push the level two formulas down in protection P_{123}, and this reduces the protected instance size by 25%. Pulling formulas up in the tree using rewriting rule in Figure 4.5(b) can have an even larger impact: P_{12} is roughly 70% smaller than P_{123} in each case. Finally, for each dataset and each protection, file sizes are presented with and without compression. We used gzip compression, which can easily be applied to the cleartext before encryption, and then after decryption by the client. The W3C Encryption Schema includes metadata to describe such pre- and post-processing of data. With gzip, under P_{123}, t' is in fact smaller than the original data. The positive impact of the rewritings above are only slightly diminished when compression is applied.

It should be emphasized that the encrypted instance generated is capable of supporting

many different access policies. In the absence of our techniques, a separate instance must be published to each user according to their access rights. Therefore a small constant factor increase in instance size is extremely favorable in many cases.

Generation and Decryption Time The graph presented in Figure 4.10 measures the generation time (above the x-axis) and decryption time (below the x-axis)[8]. The number in parentheses next to the protection name is a count of the number of keys used in the protection. The extra time to compress is more than compensated by the time saved by processing less data, so that compression actually reduces generation and decryption time overall. The two rewritings mentioned above have a modest positive impact on generation time: 12% and 18% respectively.

The absolute generation and decryption times of our preliminary implementation do not reflect the possibilities for fast processing. The symmetric algorithms used here have been implemented [7] to run orders of magnitude faster than numbers reported here. In fact, absolute throughput of optimized encryption implementations appears to far exceed the throughput of many XML parsers, so we expect that the addition of our techniques to a data processing architecture would not be the bottleneck.

4.9 Related Work

This section contains a thorough review of related research including similar controlled publishing frameworks, access control for XML, and the use of trusted computing architectures for XML data.

4.9.1 Data Encryption

It has been the classical goal of the cryptographic community to prevent the unauthorized disclosure of data communicated over networks or stored in computer systems [39, 94]. As mentioned in Section 2.4, Gifford proposes a "passive protection mechanism" [72], using cryptography to provide secrecy and authentication of data blocks or files, and granting

[8] for the Sigmod Record dataset; other results were similar.

access by sharing an appropriate key set. The key mechanisms introduced include AND, OR and general threshold schemes. The present work extends Gifford's techniques by adapting them to the hierarchical XML data model, supporting complex access control policies that may be expressed in this data model, and providing a performance analysis. Also, the emphasis of the present work is secrecy only, not authenticity.

The XML Encryption Recommendation [51] describes a format for representing encrypted data as XML. We have adopted the format for representing our encrypted data instances, and described its basic features in Section 4.4. It is not a critical aspect of the techniques described here, and could easily be replaced by an alternative format. A format for representing encrypted data does not, by itself, meet any of the goals of protected data publishing.

4.9.2 Protected data publishing

The work of Bertino and Ferrari [13, 15] precedes this work and shares the goal of controlling access to published XML data. The authors present a system for generating encrypted XML documents consistent with stated access control policies, including algorithms for evaluating access control policies on an XML document, key generation, encryption of an XML document, and the proper association of keys with user roles. In [14] they consider key management in the context of repeated updates to data.

The work presented here differs in a number of respects. First, their access control policies are limited to simple path expressions. Our access control policies are more expressive, based on XPath [30] and XQuery [18], and we provide a static analysis of our access control policies (in Section 4.6.2). While the exact physical details of their encryption are not clearly stated, the encryption is node-based. They do not address the security limitations of node-based encryption (see Section 4.5) and do not offer alternatives like nested encryption. They do not support formulas over keys, or secret-sharing techniques, which can reduce the number of keys transmitted to users, and they do not include data value keys for flexible access control. Further there is no performance analysis of any of the techniques proposed, and no consideration of the partial disclosures that result from the publication of encrypted data.

Crampton [32] considers simple access control policies expressed using XPath, and how they can be enforced using encryption (and also view selection). The access model is role based, and existing hierarchical key generation schemes are proposed for assigning keys to regions of the XML document.

Oriol and Hicks [109] propose a formalism for tagging unstructured data collections with categories for both organization and protection. The tags they apply to sets of data items include basic values, as well as surrogates for symmetric and asymmetric encryption keys. Tags can be combined with disjunction and conjunction. The tags are used to select data items and to prevent access to data items.

The use of data value keys to enforce binding patterns over XML data has been investigated by the author in [101]. That work extends techniques for publishing access controlled membership lists [63, 64], and protecting forensic databases [19]. The use of data value keys was formalized and enhanced in [103], and is included in this dissertation.

4.9.3 XML Access Control: Languages

We introduced XML access control languages in Section 2.2.2. Two of the first languages, proposed by Kudo and Hada [86] and by Damiani, et. al. [37], were the basis for a subsequent standardization effort that resulted in the OASIS Extensible Access Control Markup Language (XACML) [108]. XACML policies are themselves described as XML documents, and policy evaluation has been implemented [87]. In [86], the authors use XPath to describe the target elements, and add provisional authorization, where users are granted access on the condition that some further actions are taken (either by the user or the system). An access control language, with formal semantics described by reference to XPath, is proposed in [68] and compared nicely with other access control languages [106, 69, 37, 15, 86].

Our policy queries could be expressed as XACML policies. Unlike our formalism, XACML is not based on a query language like XQuery. Rules requiring joins, like our Policy Query 2.3 (Sec. 4.2.1) are expressed in an ad-hoc syntax.

4.9.4 XML Access Control: evaluation, optimization, and analysis

Each request for access to a protected XML document requires evaluation of the access control policy by the *security processor* which computes a boolean response of *grant* or *deny*. Evaluation may be performed with a query submitted by the user, or alternatively, the accessible view of the XML document may be materialized in its entirety, prior to query evaluation. The accessible view of an XML document may not conform to its original DTD or XML Schema, as elements may be omitted.

In [69], access policies are translated into XSLT [138] programs that compute the accessible view of the document for a given user. In [37] an algorithm is described which first labels the XML tree according to accessibility imposed by each rule, and then performs a second pass over the tree to resolve conflicts and prune. To improve the efficiency of access decisions in query processing, the authors of [142, 143] design an accessibility map which records, for each data element, the set of users permitted to access it. Locality of access in XML documents is exploited to reduce the size of the accessibility map, and to permit space and time efficient processing. Fan et al. propose security views, [62] where an access policy is defined by annotating the document DTD. Users query the accessible portions of the document by reference to a derived security view definition. The authors also propose rewriting techniques which transform queries over the accessible view into safe admissible queries that can be evaluated directly over the original XML document.

It is the goal of [106] to improve query evaluation under access policies by statically analyzing the query and access policy. Static analysis may show that a query will always refer only to accessible content, for all input documents. For such safe queries, security policy evaluation may be omitted. The analysis described in sound, but not complete: when it is not possible to determine query safety statically, policy evaluation must be performed at runtime. The authors of [29] devise optimization techniques for query evaluation under access policies. They assume an XML document labeled with access levels, and optimize evaluation by eliminating expensive tree traversals using global access information contained in the document DTD.

Yang and Li [139] address the problem of unintended disclosures that may occur even

when access control is correctly specified and enforced. The disclosures result from the fact that semantic constraints hold over the data instances. Removing data items therefore does not necessarily afford secrecy. The authors propose algorithms for computing the maximal document that can be published without allowing illegal inferences.

4.9.5 Protection using a trusted computing base

Recall that conventional access control mechanisms rely on a trusted security processor to evaluate and grant access. When data is published on the web, trusted processing is generally not available at client sites. An exception to this may be secure operating environments that can be provided by tamper resistant hardware. (See Section 4.1 for a description of secure operating environments and their comparison to the architecture considered in this chapter.)

The use of a client-side secure operating environment to protect data was investigated in [22, 21]. The target architecture is a smart card, which would perform data decryption, access control evaluation, and query processing outside the server, but in a trusted manner. To cope with severe resource limitations, the authors describe query splitting techniques which permit some computation to be performed by the server, and some by the untrusted client, reducing the processing and storage burden on the smart card. The goal of [20] is to evaluate access control policies in a resource-constrained secure environment running on a client. A security processor is described that can process streaming encrypted XML data by using an index structure containing accessibility and structural information about the data.

Chapter 5

INTEGRITY IN DATA EXCHANGE

This chapter focuses on data integrity, leaving behind the confidentiality issues addressed in the previous two chapters. First, we present a vision for managing authenticity in distributed data exchange by annotating data with cryptographic evidence of its authenticity. Then, we address a key implementation challenge we believe is critical to realizing this vision: efficiently maintaining and verifying proofs of integrity in relational databases.

5.1 Introduction

5.1.1 Vision for integrity in data exchange

Data integrity is an assurance that unauthorized parties are prevented from modifying data. Participants in distributed data exchange include primary data sources, intermediate sources, and end users (as illustrated in Figure 1.1). Integrity benefits both primary sources (who need to make sure data attributed to them is not modified) and end users (who need guarantees that the data they use has not been tampered with). After publishing data, a source can never directly prevent the modification of data by recipients, since they are autonomous and not regulated by a trusted system. However it is possible to annotate data with virtually unforgeable evidence of its authenticity that can be verified by any recipient. To do this, data sources need techniques which allow them to annotate data with claims of authenticity. These claims should be difficult to forge or transfer, and must be carried along with the data as it is exchanged and transformed. In addition, users should be able to derive useful integrity guarantees from query results containing these claims.

The ultimate goal, therefore, is to develop a framework to (1) allow authors to annotate data with evidence of authorship, (2) allow recipients to query, restructure, and integrate this data while propagating the evidence, and (3) enable recipients to derive useful conclusions about the authenticity of the data they receive. To accomplish these goals we propose two related integrity annotations which are applied to data to represent useful claims of

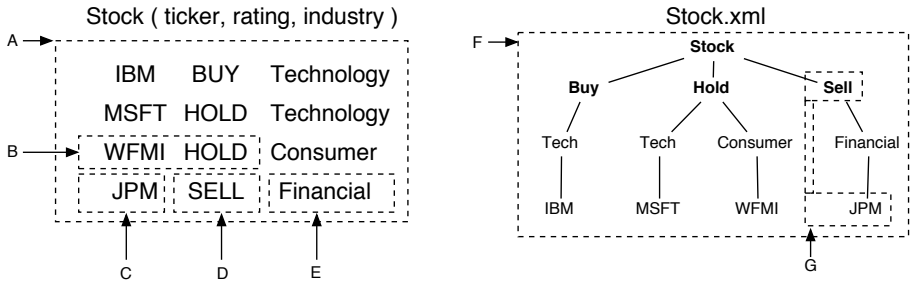

Figure 5.1: A relational table of stock recommendations (left), the same data represented as XML (right), and an illustration of fragments of the data to be signed.

origin authenticity.

5.1.2 Integrity Annotations

An annotation is a label attached to a data fragment. For relational data, the fragment may be an individual attribute, a tuple, or a set of tuples. For XML data, the fragment may be a complex subtree or a forest of subtrees. We propose below two related forms of annotation – signature and citation – which are used by data authors and consumers to represent claims of origin authenticity.

Signatures

We described digital signatures in Section 2.4. An author signs data to ensure that others cannot modify it. The granularity of signatures can vary: an author can sign an entire table, a tuple, a single column value. Often signatures are used to secure two or more data values in an unmodifiable way, as shown in the next example.

Example 5.1.1 Figure 5.1 shows stock recommendations represented as a relational table Stock(ticker, rating, industry) and as an XML document. The dotted regions illustrate portions of the data that may be signed, which we call the target fragment of a signature. Signature $sig(A)$ is applied to target A, i.e. the entire table, and $sig(F)$ is similarly applied to the entire document Stock.xml. If the user wanted to evaluate the performance of the portfolio represented by Stock, then these signatures provide integrity: the removal of

poorly performing stocks, or the addition of outperforming stocks, would be detected during signature verification. Signature $sig(B)$ and $sig(G)$ are applied to ticker-rating pairs. This associates the ticker name with the rating in an unmodifiable way. However, a collection of such signed tuples does not prevent deletions, rearrangements or additions to the collection. Signatures $sig(C)$, $sig(D)$, and $sig(E)$ are applied to individual attribute-value targets. By themselves, these three are probably not useful signatures since they do not authenticate the association between ticker and rating, which is of primary importance here.

The choice of signature granularity is application dependent. The signature of an entire database protects against all possible changes, but may be inefficient since verification must be performed on the entire database. In practice authors sometimes want to authorize smaller pieces of data. In many contexts, the author may wish to publish data signed in more than one way, with varying granularity.

Citations

We propose another integrity annotation that allows for the *citation* of signed data. We define a citation to be an annotation of a data fragment (the derived data), with a query and a reference to some signed data (the target). A citation represents a claim of authenticity: that the derived data is the result of evaluating the query on the target fragment. Equivalently, the derived fragment is the answer of a given view, evaluated over the signed target data. The following examples provide some intuition, and an illustration of the flexibility of citations.

Example 5.1.2 Consider again the stock recommendations in Figure 5.1. Each row of Table 5.1 describes the components of a citation. The first column is the derived fragment (tuples or sets of tuples in this case) to which the citation refers. The third column names the target fragment, where A refers to the data signed by $sig(A)$. The second column is the citation query which relates these two. It is expressed as a conjunctive query over the signed target fragment, and its result is the target fragment. The last column indicates that the target is backed by a signature. The intended meaning of each citation is described below:

1. Citation (1) has derived data consisting of two tuples (IBM, BUY) and (MSFT, HOLD). The citation claims that these two tuples are the result of evaluating query C_1 on the target (the entire Stock table), which is signed by signature $sig(A)$. That is, the claim says that these tuples (and only these tuples) make up the ticker and rating fields of stocks in the Technology sector recommended by the signing author Alice.

2. Citation (2) is very similar, but with a different condition in the citation query (it selects stocks in the Consumer sector).

3. Citation (3) consists of the same derived data (WFMI, HOLD) as example (2), however its citation query is different: it claims that the derived data is *contained* in the result of query C_3. In this case, the claim is that (WFMI, HOLD) is one of the stocks rated HOLD in the original signed table, but that other qualifying tuples may have been omitted (in fact MSFT has been omitted). If this claim is verified, the user will know that spurious tuples have not been added to the result, but will not be sure that some tuples have not been removed. (In Section 5.1.4 we will define this as a guarantee of correctness, without a guarantee of completeness.)

4. In each of the citations above, the target data was signed. In other cases the target data may instead be cited, resulting in the composition of citations. The derived fragment from Citation (1) is T_1. Citation (4) therefore refers to a fragment that is itself cited. The claim of authenticity here is the composition of the individual claims. That is, the citation claims that tuple (MSFT, HOLD) is the result of query C_4 evaluated on table T_1 which itself is the result of citation query C_1 on the original data signed with $sig(A)$. This results in a claim that (MSFT, HOLD) is the ticker and rating of all Technology stocks with a HOLD recommendation in the original signed table.

Citations are useful because they do not require the compliance of the author, and because they provide additional flexibility if the signature on the source data does not permit the extraction a user desires. For example, as the author, Alice may choose to sign

Table 5.1: Citations, referring to the relational data in Figure 5.1.

	Derived fragment	Citation query	Target fragment	Signature / Citation
$T_1 =$	(IBM, BUY) (MSFT, HOLD)	$C_1(t, r)$:- A(t, r, "Technology")	A	Sig A
$T_2 =$	(WFMI, HOLD)	$C_2(t, r)$:- A(t, r, "Consumer")	A	Sig A
$T_3 =$	(WFMI, HOLD)	$C_3(t, r) \subseteq$ A(t, "HOLD", i)	A	Sig A
$T_4 =$	(MSFT, HOLD)	$C_4(t, r)$:- T_1(t, "HOLD")	T_1	Cit T_1

(only) the stock table in its entirety. She may not want to sign individual tuples in the stock table because she doesn't want to authenticate individual tuples in isolation. Another user, Bob, may only be interested in replicating the tuple (WFMI,HOLD) from Alice's table for use in a derived database. This individual tuple can't be published with Alice's signature (Alice didn't sign it, and Bob shouldn't be able to forge her signature). But it can be cited by Bob, using Alice's data as a target.

Signatures can only be generated by the author, but citations can be generated by any downstream user of the data. Even so, citations may not offer the same level of integrity guarantee as a signature, and as the example shows, the same data may be cited using more than one citation query resulting in different authenticity conditions. Notice that citations (2) and (3), as well as $Sig(B)$ from Example 5.1.1, are each annotations representing a claim of integrity about the same data fragment: (WFMI, HOLD). Each of these integrity annotations has a different meaning, and in some cases the distinction could be important.

Note that a citation is a generalization of a signature. Consider a citation whose query is the identity query. The citation says that the derived fragment is precisely the target fragment, which is backed by some signature. This is equivalent to annotating the derived fragment with the signature of the signed target fragment. Also, a citation reports on the provenance of the cited data, describing it's relationship to the source data. A discussion of related work on data provenance is provided in Section 5.7.

Since a citation is merely a claim, it must be verified by checking the signature of the cited source, and verifying that the cite fragment is in fact the result of the citation query evaluated on the citation source. Designing an efficient verification mechanism for citations is the focus of this chapter.

5.1.3 Using hash trees to verify citations

Note that there is a naive strategy for verification of a citation, which is to retrieve the original signed target data, compute the citation query and compare the result with the annotated data. This may be inefficient or impossible in a data exchange setting, and our goal is to avoid this. Research into consistent query protocols [82, 110, 47, 45] can provide a more efficient verification process for a certain limited class of queries. The techniques are based on Merkle hash trees [95, 96] and allow signing of a database D such that a qualifying query Q and a possible answer x to Q, a verification object can be constructed which proves that $x = Q(D)$. This matches the semantics of citations described above. The verification object may be constructed (with knowledge of the query) by either the data owner or an untrusted third-party with access to the signed database. The important property is that the verification guarantees consistency with the original signed data, and that third-parties cannot forge verification objects.

5.1.4 Using hash trees for integrity in database systems

Hash trees are described in detail in Section 5.2. The efficient implementation of hash trees in relational databases also has applications to conventional client-server databases. In database systems, integrity is provided by user authentication and access control. Users are required to authenticate themselves and are limited in the operations they can perform on columns, tables, or views. Unfortunately, in real-world systems these mechanisms are not sufficient to guarantee data integrity. The integrity vulnerabilities in a modern database system stem from the complexity and variety of security-sensitive components, integration of database systems with application level code, underlying operating system vulnerabilities, and the existence of privileged parties, among others. Consequently, when a data owner uses a database to store and query data, she is forced to trust that the system was configured

properly, operates properly, and that privileged parties and other users behave appropriately. Hash trees are a safeguard against these vulnerabilities because they provide the author of data with a mechanism for detecting tampering. When verification succeeds, the data owner has a strong guarantee that none of the vulnerabilities above has resulted in a modification in their data.

The use of hash trees in both data exchange and client-server databases is illustrated in Figure 5.2. In the top figure, Alice is a client for an untrusted database server which stores her data and processes queries on her behalf. When she queries the database, she performs a verification procedure to check that no modifications have been made to her data.

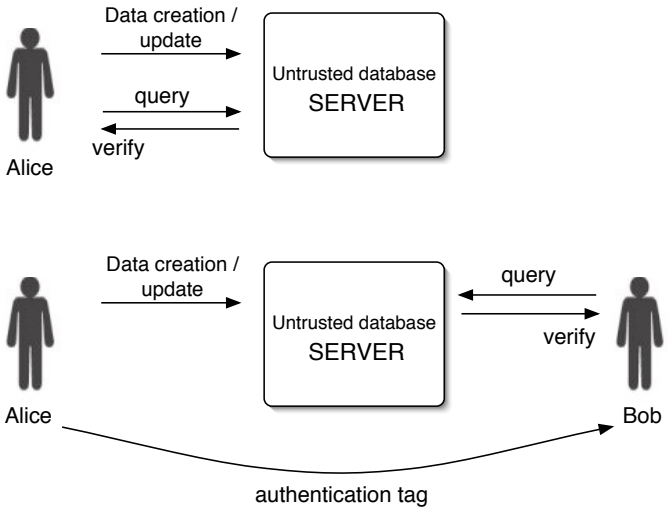

Figure 5.2: Alternative uses for integrity mechanisms.

The lower figure illustrates distributed data exchange. Alice acts as a source and data owner, publishing data for storage on an untrusted system. Bob acts as recipient of the data and would like to verify the claim made by a citation. Bob retrieves data from the untrusted server, and also retrieves an authentic copy of the hash tree root for verification. We describe these operations in more detail below.

We focus initially on a single author who is the sole individual with rights to create and modify data. Many parties may issue queries, but integrity is judged with respect to the

author. (We discuss extensions of our results to the multiparty case in Section 5.6.) To formalize integrity guarantees, we assume the author creates a relation R, stores it at the server (and not locally), and asks queries over the database. We further assume that the author updates the relation and that the sequential authentic states of the database are $R_1, R_2, \ldots R_n$. When a user asks a (monotone) query q over R, they receive ans. The key integrity properties we consider are:

Correctness Every tuple was created or modified only by the author:

$$ans \subseteq \bigcup_{1 \leq i \leq n} q(R_i)$$

The union on the right consists of all tuples present in any state of the database. Correctness asserts that the answer set is composed of only these tuples.

Completeness No qualifying tuples are omitted from the query result: for some i, $q(R_i) \subseteq ans$.

Consistency Every tuple in the database or query result is current; i.e. it is not possible for the server to present as current a tuple that has been been removed, or to mix collections of data that existed at different points in the evolution of the database. Consistency means $ans \subseteq q(R_i)$ for some i.

If the author were to sign tuples individually, verification by the client would prove correctness only. Consistency is a stronger condition which implies correctness. The hash tree techniques to be described provide consistency (and thus correctness) upon verification. In some cases hash trees can provide completeness as well. The index i, which identifies particular states of the database, corresponds to a version of the authentication tag which is changed by the author with updates to the database. Proofs of consistency or completeness are relative to a version of the authentication tag.

Note that the database server is largely oblivious to the fact that Alice is taking special steps to ensure the integrity of her data. The server stores a modified schema for the database which, in addition to the base data, also includes integrity metadata. The integrity

metadata consists of one or more specially-designed tables each representing a hash tree [95, 96, 47]. Alice's database queries are rewritten by the middleware to retrieve the query answer along with some integrity metadata. The middleware performs an efficient verification procedure, returning to Alice the query answer along with notice of verification success or failure.

5.1.5 Contributions and chapter organization

The goal of this chapter is to allow a user to leverage a small amount of trusted client-side computation to achieve guarantees of integrity when interacting with a potentially vulnerable database server. The remainder of the chapter presents the design, implementation and performance evaluation of hash trees for use with a client-server relational database. We describe a novel relational representation of a hash tree, along with client and server execution strategies, and show that the *cost of integrity* is modest in terms of computational overhead as well as communication overhead. Using our techniques we are able to provide strong integrity guarantees and process queries at a rate between 4 and 7 times slower than the baseline, while inserts are between 8 and 11 times slower. This constitutes a dramatic improvement over conceivable methods of ensuring integrity using tuple-level digital signatures, and also a substantial improvement over naive implementations of hash trees in a database. Since our techniques can easily augment any database system, we believe these techniques could have wide application.

The next section presents background on Merkle hash trees, followed in Section 5.3 by the design of our relational hash trees. Section 5.4 describes optimizations, and Section 5.5 is a thorough performance evaluation. Related work is reviewed in Section 5.7 and extensions are discussed in Section 5.6.

5.2 Background

5.2.1 Enforcing integrity with hash trees

In this subsection we review the use of hash trees for authenticating relations [47, 95] by illustrating a simple hash tree built over 8 tuples from a relation R(score,name). We denote by f a collision-resistant cryptographic hash function (for which it is computationally-

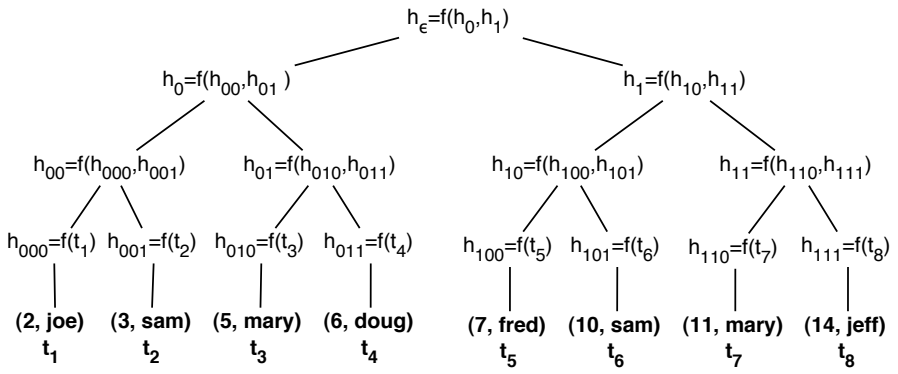

Figure 5.3: Hash tree built over a relation containing tuples $t_1, t_2, ...t_8$. f is a cryptographic hash function; comma denotes concatenation when it appears in the argument of f.

infeasible to find inputs x and x' such that $f(x) = f(x')$). We build the tree of hash values, shown in Fig. 5.3, as follows. First we compute the hash for each tuple t_i of the relation by hashing the concatenated byte representation of each attribute in the tuple. Then, to generate a (binary) hash tree, we pair these values, computing f on their concatenation and storing it as the parent. We continue bottom-up, pairing values and hashing their combination until a root hash value h_ϵ is formed. The root hash value, h_ϵ, is a short sequence of bytes that depends on each tuple in the database and on a chosen order for the tuples. (The value of the root hash can therefore be used to uniquely identify states of the database, as per the definition of consistency above.)

The computation of a hash tree uses the *public* hash function f and is deterministic, so for a given tree shape it can be repeated by anyone. Alice chooses an order for the tuples in her relation, computes the root hash h_ϵ and stores it locally and securely. She will then store the relation at the vulnerable server. In Fig. 5.3 the hash tree is perfectly balanced, but this is not required.

5.2.2 Verifying query results

The client verifies a query by checking that the query result is consistent with the root hash. To do so, the client must duplicate the sequence of hash computations beginning with the

query result, and verify that it ends with the root hash. The tree structure allows the client to perform this computation without recovering the entire relation from the server. For a set of result tuples, the nodes in the tree the server must return are those on the *hash path*, which consist of all siblings of nodes on a path from a result tuple to the root. Successful verification proves integrity under the assumption that it is impossible for the server to find a collision in the hash function f. We illustrate with the example queries below which refer to the database R in Fig. 5.3 and we assume score is a key for R.

Example 5.2.1 Select tuples with score=5. The server returns the answer tuple, t_3. The client can compute h_{010}. In order to complete the computation to the root, the client needs more values from the database, or some nodes internal to the tree. The server returns in addition the hash path consisting of nodes h_{011}, h_{00} and h_1. From these values the client can compute up the tree a new root hash h'_ϵ. Verification succeeds if h'_ϵ equals the root hash h_ϵ stored at the client. Unless the server can find a collision in the hash function, this proves that tuple t_3 is an authentic element of the database, proving consistency and correctness of the query answer.

Example 5.2.2 Select tuples with score=8. Since 8 is not present in the database, the query result is empty and correctness holds trivially. To show completeness, i.e. that there are no tuples with score=8 omitted illegally, the server must return the predecessor tuple, t_5, and the successor, t_6, and the hash path $\{h_{11}, h_0\}$.

Example 5.2.3 Select tuples with *score* between 4 and 6. The server will return answer tuples t_3 and t_4 along with their hash path $\{h_{00}, h_1\}$ which allows the client to verify correctness and consistency, as above. However there could exist other tuples in the collection matching the search condition, *score* between 4 and 6. Evidence that the answer is in fact complete relies on the fact that the tree is built over sorted tuples. The server provides the next-smallest and next-largest items for the result set along with their hash path in the tree. To prove completeness, the result will consist of t_2, t_3, t_4, t_5 and the hash path is h_{000}, h_{101}, h_{11}.

Example 5.2.4 Select tuple with name='Mary'. This query is a selection condition on the B attribute, which is not used as the sort key for this hash tree. The server may return the entire result set $\{t_3, t_7\}$ along with the hash path nodes $\{h_{011}, h_{00}, h_{111}, h_{10}\}$, however in this case only consistency and correctness is proven. The server could omit a tuple, returning for example just t_3 as an answer along with its verifiable hash path. The author will not be able to detect the omission in this case.

5.2.3 Modifying data: Insertions, deletions, updates

The modification of any tuple in the database changes the root hash, upon which the verification procedure depends. Therefore, the client must perform re-computation of the root hash locally for any insertion, deletion, or update. We illustrate with the following example:

Example 5.2.5 Insert tuple (12, jack). This new tuple t' will be placed in sorted order between tuples t_7 and t_8. More than one tree shape is possible, but one alternative is to make t' a sibling of t_7 and set $h_{110} = f(t_7, t')$. Since the value of h_{110} has changed, the hashes on the path to the root must be updated, namely h_{11}, h_1 and h_ϵ.

It is critical that the root hash be computed and maintained by a trusted party, and retrieved securely when used during client verification. If the server could compute or update the value of the root hash, it could perform unauthorized modifications of the database without violating the verification procedure. The root hash can be stored securely by the client, or it can be stored by the server if it is digitally signed. Since the root hash changes with any modification of the database, the latter requires re-signing of the root hash with each update operation.

5.3 The relational hash tree

In this section we describe techniques for implementing a hash tree in a relational database system.

5.3.1 Overview of design choices

The simplest representation of a hash tree as a relation would represent nodes of the tree as tuples with attributes for parent, right-child, and left-child. With each tuple uniquely identified by its node id, the parent and child fields would contain node id's simulating pointers. There are a number of drawbacks of this simple organization. The first is that in order to guarantee completeness the server must return not just the result set, but the preceding and following elements in the sorted order (as in Ex. 5.2.3). Second, traversing the tree requires an iterative query procedure which progresses up the tree by following a sequence of node-ids, and gathers the nodes on the hash path. Finally, the performance of this scheme depends on the tree being balanced which must be maintained upon modification of the database. Even assuming perfect balancing, experiments indicate that the time at the server to execute a simple selection query using this organization is about 12 ms (a factor of 20 times higher than a standard query) and that query times scale linearly or worse with the size of the result.

To simplify the hash path computation, a basic optimization is to store the child hashes with their parent. Then whenever the parent is retrieved, no further database access is required for gathering the children. To simplify the identification of preceeding and following elements, we translate our sorted dataset into intervals. The leaves of the hash tree are intervals, and we always search for the containing interval of a point, or the intersecting intervals of a range. This provides completeness as in Ex. 5.2.2 and 5.2.3. To remove the iterative procedure on the path to the root, we store intervals in the internal nodes of the tree representing the minimum and maximum of the interval boundaries contained in that node's descendants. The result is a relational representation of a hash tree with the only remaining challenges being (i) implementing interval queries efficiently and (ii) keeping the tree balanced.

Interval queries are not efficiently supported by standard indexes. One of the best methods for fast interval queries was presented in [112]. These techniques are a relational adaptation of an interval tree [52] and themselves require representing a tree as relations. Since any tree shape can work as a hash tree, the innovation here is to adapt interval trees

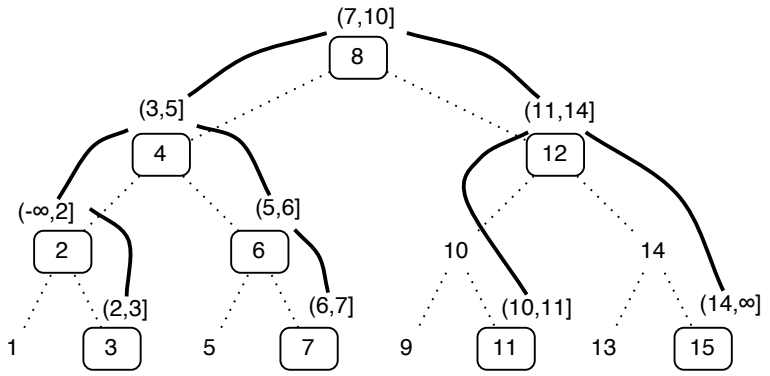

Figure 5.4: The *domain tree* T_4 is the complete binary tree, rooted at 8, shown with dotted edges. The *value tree* is represented by circled nodes and solid edges, for intervals derived from $\mathbf{adom} = \{2, 3, 5, 6, 7, 10, 11, 14\}$. The intervals shown in the tree are stored in the Auth table.

to the design of a hash tree, combining both trees into one structure. This serves dual purposes: in addition to supporting very efficient hash path queries, it also maintains a balanced tree for many data distributions.

5.3.2 Interval Trees

An interval tree [52, 112] is a data structure designed to store a set of intervals and efficiently support intersection queries. We review this data structure here, and then adapt it to a hash tree in the next subsection.

The *domain tree* T_k for positive[1] domain $\mathbf{dom} = [0..2^k - 1]$ is a complete binary tree whose nodes are labeled with the elements of $\mathbf{dom} - \{0\} = (0..2^k - 1]$. Its structure is precisely that of a perfectly balanced search tree built over the entire domain. The root of T_k is the midpoint of $\mathbf{dom} - \{0\}$, 2^{k-1}, or in bits 1000...0. Its children are 01000...0 and 1100...0. The tree has height $k - 1$ and its leaves are the odd elements of **dom**. Domain tree T_4 is shown in Fig. 5.4. We often refer to the nodes of T_k by their labels in **dom**.

Each node in T_k is the midpoint of its spanning interval, denoted $span(n)$, defined to be

[1]For illustration purposes only, we describe a relational hash tree over a positive domain, although it is easily generalized to signed integers, floats, or strings. We implemented a signed integer domain with $k = 32$.

the interval containing every element in the subtree rooted at n, including itself. The span of the root is therefore $(0..2^k - 1]$; the span of node 2, shown in Fig. 5.4 is $(0, 3] = \{1, 2, 3\}$. For a given **dom**, we denote $-\infty = 0$, and $\infty = 2^k - 1$.

An interval with endpoints in **dom** is stored at a unique node in the domain tree as defined by its *fork node*. The fork is the lowest node n such that $span(n)$ contains the interval.

Definition 5.3.1 (Fork node of interval) *Let $I = (x, y]$ be an interval with $x, y \in$ **dom** and $x < y$. The fork node fork(I) is the unique node $n \in T_k$ such that:*
(i) $I \subseteq span(n)$, and
(ii) for all descendants n' of n, $I \not\subseteq span(n')$.
It follows immediately that $n \in I$, and that n is the highest node in the domain tree such that $n \in I$.

As an example, the fork node in T_4 of interval $(3, 5]$ is 4 as shown in Fig 5.4. The computation of the fork for interval $I = (x, y]$ can be performed efficiently using bitwise operations on x and y. Since $x < y$, x and y can be written as $x = z0x_0$ and $y = z1y_0$ for (possibly empty) bit strings z, x_0, y_0. Then fork$(I) = z10^{|x_0|}$. If $x = 3 = 0011$ and $y = 5 = 0101$ then $z = 0$ and fork$(I) = 0100 = 4$.

5.3.3 Disjoint interval trees

We now adapt interval trees to our setting. Let **adom** $= \{x_1 \dots x_n\}$ be a set of data values from **dom**. We always assume the data values are different from $-\infty$ and ∞. Then they partition **dom** into a set U of $n + 1$ intervals:

$$I_0 = (-\infty, x_1], \quad I_1 = (x_1, x_2] \quad \dots \quad I_n = (x_n, \infty]$$

for $-\infty = 0$ and $\infty = 2^k - 1$. Since the elements of **adom** are distinct, we have $x_i < x_{i+1}$ for all intervals I_i. Each interval of U is stored at its fork node in the domain tree. We show next how the intervals can be connected to form a tree which overlays the domain tree.

We say a node w in the domain tree is *occupied* if there is an interval in U whose fork is w. The occupied nodes in Fig 5.4 are circled and labeled with the intervals that occupy them. Recall that the intervals of U are always non-overlapping and cover the entire domain. (Ours is therefore a special case of the structure considered in [52, 85] which is designed

to accommodate a general set of intervals.) Each node in the domain tree holds at most one interval: since a fork node n is an element of any interval whose fork is n, two disjoint intervals cannot have the same fork. Further, the root of T_k is always occupied. This is because the interval set U partitions the entire domain, so some interval in U must contain the root element 2^{k-1} and this interval's fork can only be the root of T_k. We show next that for any disjoint interval set U as above, the occupied nodes can always be linked to form a unique (possibly incomplete) binary tree, called the *value tree* for U. Let SUBTREE(x) in T_k be the subtree rooted at x.

Property 5.3.2 *For any node x in domain tree T_k, if there is at least one occupied node in* SUBTREE(x)*, then there is always a unique occupied node y which is the ancestor of all occupied nodes in* SUBTREE(x)*.*

Proof: (Sketch) This property follows from the fact that if y and y' are two occupied nodes in SUBTREE(x) then their least common ancestor must be occupied. If y is the ancestor of y', or vice versa, this is trivially true. Otherwise, suppose that their least common ancestor is a distinct node z that is not occupied. Let I be the interval occupying fork node y. Since y is a descendant of z, and fork(I) = y, the element z cannot be contained in I. The same argument shows that z cannot be contained in I'. However, z must be present in some interval I_0, since the intervals partition the domain. Interval I_0 must occupy node z. It's fork cannot be lower in the tree because it contains z. It cannot be higher in the domain tree either: the fact that it has occupied nodes beneath it implies that I_0 does not contain any ancestor of z. □

As an example, in Fig 5.4, the unique occupied ancestor in SUBTREE(10) is node 11. Property 5.3.2 implies that nodes 9 and 11 could never be occupied while 10 remains unoccupied.

Definition 5.3.3 (Value tree) *For a set of disjoint intervals U derived from* **adom** *as above, and domain tree T_k, the value tree V_U is a binary tree consisting of the occupied nodes from T_k and defined inductively as follows:*

- *root*(V_I) = *root*(T_k)
- *For any value node x in V_U, its right (resp. left) child is the unique occupied ancestor of the right (resp. left) subtree of x in T_k, if it exists.*

x is a leaf in V_U iff the right and left subtrees of a value node x are both unoccupied.

In Fig. 5.4 the value tree is illustrated with solid edges connecting the occupied nodes of T_4.

Benefits of the value tree

In summary, the domain tree is static, determined by the domain, while the value tree depends on the set of values from which the intervals are derived. The value tree has a number of important properties for our implementation. First, by design, it is an efficient search structure for evaluating range intersection queries (i.e. return all stored intervals that intersect a given interval $I = (x, y]$). Such a query is evaluated by beginning at the root of the value tree, traversing the path in the value tree towards $fork(I)$ and checking for overlapping stored intervals. Secondly, it provides an organization of the data into a tree which we use as the basis of our hash tree. This avoids explicit balancing operations required by other techniques. Finally, the relationship between the domain tree and value tree allows us to avoid expensive traversals of the tree at the server. Instead of traversing the path from a node in the value tree to the root, we statically compute a *superset* of these nodes by calculating the path in the domain tree. We then probe the database for the value tree nodes that exist. The sets defined below are used for query evaluation in Sec. 5.3.5, and compute all the forks of nodes in the value tree necessary for client verification.

Definition 5.3.4 (Ancestors) *For $x \in$* **dom**, $\mathsf{Ancestors}(x)$ *is the subset of* **dom** *consisting of all domain tree nodes whose span contains x:*

$$\mathsf{Ancestors}(x) = \{n \mid x \in span(n)\}$$

Definition 5.3.5 (Range Ancestors) *For interval $(x, y]$,* $\mathsf{rangeAncestors}(x, y)$ *is the subset of* **dom** *consisting of all domain tree nodes whose span intersects $(x, y]$:*

$$\mathsf{rangeAncestors}(x, y) = \{n \mid (x, y] \cap span(n) \neq \emptyset\}$$

$\mathsf{Ancestors}(x)$ includes all fork nodes where an interval containing x *could* be stored. $\mathsf{rangeAncestors}(x, y)$ includes all fork nodes where an interval containing $(x, y]$ *could* be stored. Note that these sets do not depend on the current state of the database, only the domain. For our example domain, $\mathsf{Ancestors}(13) = \{8, 12, 14, 13\}$ and range-$\mathsf{Ancestors}(6,9)$ $= \{8, 4, 6, 7, 12, 10, 9\}$.

Properties of a disjoint interval tree

Each set **adom** determines a unique value tree. If **adom** is empty, then the resulting interval set contains only one interval $(-\infty, \infty]$, and the value tree consists of a single node: the root of T_k. At the other extreme, if $\mathbf{adom} = \mathbf{dom} - \{-\infty, \infty\}$ then every node of the domain tree will be occupied, and the value tree will be equal to the domain tree. We describe next some properties of the value tree which are implied by sets **adom** derived according to a known distribution.

The depth of a node $n \in T_k$ is the number of edges along the path from the root to n (hence 0 for the root). The width of an interval determines the maximum depth of the interval's fork.

Property 5.3.6 (Maximum depth of interval) *Let $I = (x, x']$ be an interval and define $j = \lfloor \log_2(x' - x) \rfloor$. Then the depth of fork(i) is less than $k - j$.*

Proof: (Sketch) If n has depth j, the $span(n)$ is an interval of width $2^{k-j} - 1$, since the root spans the whole domain, and moving from a node to its child divides the span in half. It follows from the definition of fork that $span(n)$ is the largest interval that can be stored at the node n, since any lower node could never contain it. The value of j is defined to be $\lfloor \log(x' - x) \rfloor$, so that the width of the interval is $2^j \leq (x' - x) \leq 2^{j+1}$. □

This result implies that if **adom** consists of values spread uniformly, then the value tree fills T_k completely from the top down. Alternatively, if **adom** consists of consecutive values, then Prop 5.3.2 implies that the value tree fills T_k completely from the bottom up. These observations allow us to prove:

Property 5.3.7 (Value tree shape) *Let* **adom** *be a set of $2^m - 1$ elements (for $m < k$). If (1)* **adom** *is spread uniformly or (2) it consists of consecutive values, then the value tree has height m.*

Proof: (Sketch) Since the elements of **adom** are uniformly distributed, the expected width of each resulting interval will be 2^{k-m}. Property 5.3.6 implies that the occupied nodes each have depth less than m. T_k consists of precisely $2^m - 1$ nodes at a height less than m, so it follows that every node of T_k must be occupied, and the value tree is complete. □

The relevant issue for our implementation is the length of paths in the value tree. Both a uniform distribution and a set of consecutive values result in a minimal height of the value tree. The question of the worst case distribution in this setting remains open. We return to the impact of the data distribution, and domain size, k, in Sec. 5.5.

5.3.4 Relational hash tree

In this subsection we describe the relational representation of a hash tree based on interval trees. Figure 5.5 contains table definitions, indexes, and user-defined functions, while Figure 5.6 contains the important verification queries. Given a relation $R(A, B_1, ...B_m)$, we choose the sort attribute of the hash tree to be A and assume the domain of A can be represented using k bits. A is not necessarily a key for R, and we let $\mathbf{adom}_A$ be the set of distinct A-values occurring in R. We form disjoint intervals from this domain as described above, build a disjoint interval tree, and encode each node in the *value tree* as a tuple in table $\mathsf{Auth}_{R.A}$. The table Data_R stores each tuple of the original table R, with an added field fork which is a foreign key referencing $\mathsf{Auth}_{R.A}$.fork. We drop the subscripts for Auth and Data when the meaning is clear. For tuple $t \in$ Auth, t.fork is the fork of the interval (t.predA,t.A]. Hash values for the left child and right child of the node are stored in each tuple. In addition, the hash of the node *content* is stored in attribute hashed-content. The *content* of a value tree node is a serialization of the pair (pred,A), concatenated with a serialized representation of all tuples in Data agreeing on attribute A, sorted on a key for the remaining attributes. The hash value at any node in the value tree is computed as f(Lhash, hashed-content, Rhash). For internal nodes of the value tree, Lhash and Rhash are hash values of right and left children of the node. For leaf nodes, Lhash and Rhash are set to a public initialization value. All hash values are 20 bytes, the output of the SHA-1 hash function. Note that in a conventional hash tree, data values only occur at the leaves while in our interval hash tree, data values are represented in all nodes.

5.3.5 Authenticated query processing

The client-server protocol for authenticated query and update processing is illustrated in Fig. 5.7. We describe next server query processing, client verification, and updates to the

TABLES
Auth$_R$(<u>fork</u> bit(k), predA bit(k), A bit(k), Lhash byte(20), hashed-content byte(20),
Rhash byte(20))
Data$_R$(fork bit(k), A bit(k), B_1 ... B_m)

INDEXES
Auth-index CLUSTERED INDEX on (Auth.fork)
Data-index INDEX on (Data.fork)
⟨additional user indexes on Data not shown⟩

FUNCTIONS
Ancestors(x bit(x)) ⟨defined in Sec. 5.3.3⟩
rangeAncestors(x bit(x),y bit(x))

Figure 5.5: Table, index, and function definitions for the relational hash tree implementation.

(Q1) Selection query on A: R.A = $x

```
SELECT Auth.*
FROM Ancestors($x) as F, Auth
WHERE F.fork = Auth.fork
ORDER BY F.fork
```

(Q2) Range query on A: $x < R.A ≤ $y

```
SELECT Auth.*
FROM rangeAncestors($x,$y) as F, Auth
WHERE F.fork = Auth.fork
ORDER BY F.fork
```

(Q3) Arbitrary query condition: cond(R.A, R.B1 .. R.Bm)

```
SELECT Auth.*
FROM Auth, Data, Ancestors(Data.A) as F
WHERE F.fork = Auth.fork AND cond(Data.A,Data.B1 .. Data.Bm)
ORDER BY F.fork
```

Figure 5.6: Query definitions for the relational hash tree implementation.

database.

Server query processing

The query expressions executed at the server are shown in Figure 5.6. For selection and range queries on the sort attribute, $Q1$ and $Q2$, and arbitrary query conditions, $Q3$. They each retrieve from Auth the result tuples along with paths to the root in the value tree. We avoid iterative traversal in the value tree by computing the sets Ancestors(x) or rangeAncestors(x, y) and performing a semijoin. Note that the computation of the ancestors makes no database accesses. It is performed efficiently as an user-defined procedure returning a unary table consisting of nodes from the domain tree. The following examples illustrate the execution of $Q1$ and $Q2$.

Example 5.3.8 SELECT * FROM R WHERE R.A = 13 Referring to Fig. 5.4, we compute Ancestors(13) which is equal to $\{8, 12, 14, 13\}$. $Q1$ joins these nodes with the value tree nodes in Auth_R. The result is just two tuples representing nodes 12 and 8 in the value tree. Node 12 holds interval $(11, 14]$ which contains the search key 13, proving the answer empty but complete, and node 8 is included since it is on the path to the root.

Example 5.3.9 SELECT * FROM R WHERE 6 < A ≤ 9 Computing rangeAncestors$(6, 9)$ yields the set $\{8, 4, 6, 7, 12, 10, 9\}$ since each of these nodes in the domain tree has a span intersecting $(6, 9]$. Of these, only nodes $\{8, 4, 6, 7, 12\}$ are in the value tree, and are retrieved from the database. Note that some intervals stored at these nodes do not overlap the query range $(6, 9]$ (for example, $(3, 5]$ is retrieved with node 4). Nevertheless, node 4 is required for reconstructing this portion of the value tree since nodes 6 and 7 are its descendants. The client will perform a final elimination step in the process of verifying the answer.

Arbitrary queries

For queries that include conditions on attributes $B_1, ...B_m$ the interval hash tree cannot be used to prove completeness of the query answer, but can still be used to prove correctness and consistency. Any complex condition on $B_1, ...B_m$ can be evaluated on Data, resulting in a set of values for attribute A. These values will be distributed arbitrarily across the

domain and therefore arbitrarily in the value tree. They are fed into the Ancestors function to retrieve all paths up to the root. Duplicates are eliminated, and then these nodes are joined with Auth (as in queries $Q1$ and $Q2$). The resulting query, called an *arbitrary condition query* is shown in Figure 5.6 as $Q3$.

Example 5.3.10 The query in Ex. 5.2.4 asked for all tuples from R with *name*='Mary'. The query result consists of tuples with *scores* 5 and 11. To authenticate this query, the fork ancestor set is computed to be $\mathsf{Ancestors}(5) \cup \mathsf{Ancestors}(11) = \{8, 4, 6, 5\} \cup \{8, 12, 10, 11\}$ $= \{8, 4, 6, 5, 12, 10, 11\}$.

Execution plan

For queries Q_1, Q_2 and Q_3 the favored execution strategy is to materialize the Ancestors table, and perform an index-nested loops join using the index on Auth_R. The query optimizers in the two database systems we tried chose this execution plan. For Q_1, the Ancestors set is quite small – it is bounded by parameter k of the domain tree (32 for most of our experiments). Thus evaluation of Q_1 consists of not more than k probes of the index on Auth. For range queries, the Ancestors set is bounded by the result size plus $2k$. The execution of range queries can be improved by utilizing the clustering of the index on Auth. This optimization is described in Sec. 5.4.3.

Client verification

In each case above, the server returns a subset of tuples from Auth which represent a portion of the value tree. The client verifies the query answer by reassembling the value tree and recomputing hashes up the tree until a root hash h'_ϵ is computed. To enable the client to efficiently rebuild the tree, the result tuples can be sorted by the server.

Insertion, deletion, and update

Updating any tuple in the database requires maintenance of the interval hash tree, namely addition or deletion of nodes in the value tree, and re-computation of hashes along the path to the root from any modified node. These maintenance operations are integrity-sensitive, and can only be performed by the client. Therefore, before issuing an update, the client

must issue a query to retrieve the relevant portions of the hash tree, verify authenticity, and then compute the inserts or updates to the database. The insertion protocol between client and server is illustrated in Fig. 5.7. We focus for simplicity on inserts (deletes are similar, and updates are implemented as deletes followed by insertions).

Example 5.3.11 To insert a new tuple (13, ..) the client issues the selection query for $A = 13$. This is precisely the query described in Ex. 5.3.8, and retrieves nodes 8 and 12 from the value tree. Node 12 contains interval $(11, 14]$ which must be split, with the insertion of 13, into two intervals: $(11, 13]$ and $(13, 14]$. This requires an update of the tuple representing value tree node 12, changing its interval upper bound from 14 to 13. Interval $(13, 14]$ will be stored at the formerly unoccupied node 14, which requires insertion of a new tuple in Auth representing the new value node. The hashes are recomputed up the tree from the lowest modified node in the value tree.

In general, executing an authenticated insert involves the cost of an authenticated query, the insertion of one new Auth tuple, and updates to h Auth tuples, where h is the depth of the fork of the new interval created by the insertion. Although each update to Auth can be executed efficiently using the index on node, the large number of updates can cause a severe penalty. We address this problem next by bundling the nodes of the value tree.

5.4 Optimizations

5.4.1 Bundling nodes of the value tree

The execution cost of queries and inserts depends on the length of paths in the value tree, which is determined by the data distribution and bounded by the maximum depth in the tree k. Reducing the length of paths in the value tree reduces the number of index probes executed for queries, and reduces the number of tuples modified for insertions. To reduce path length, we propose grouping nodes of the value tree into bundles, and storing the bundles as tuples in the database. For example, we can imagine merging the top three nodes $(8, 4, 12)$ in the tree of Fig. 5.4 into one node. We measure the degree of bundling by the height of bundles, where a bundle of height b can hold at most $2^b - 1$ value tree nodes ($b = 1$ is no bundling). The schema of the Auth table is modified so that a single

tuple can hold the intervals and hashes for $2^b - 1$ nodes in the value tree. We use variable-length binary large objects to store collections of hashes and intervals for each bundle. The resulting schema is: Auth(fork int, intervals varbinary, hashes varbinary) where intervals encodes a list of intervals, and hashes encodes a list of corresponding left, right, and content hashes which are decoded at the client. Finally, the Ancestors and rangeAncestors functions are generalized to account for bundling.

5.4.2 Inlining the fork ancestors

Although the Ancestors set is efficiently computed for selection queries on the sort attribute, our experiments show that when evaluating arbitrary query conditions that return a large number of distinct A-values, computation of the Ancestors set can be a prohibitive cost. To remedy this, we have proposed trading off space for computation, and inlining the ancestor values for each tuple in Data. This requires that the schema of the Data table be modified to accommodate $\lceil k/b \rceil$ fork nodes, where k is the parameter of the domain tree and b is the bundling factor. For example, for $b = 4$ we store 8 fork nodes fork1, fork2, ... fork8 with each tuple in Data and we would rewrite query $Q3$ (from Figure 5.6) as shown below, using the operator combine to coalesce the multiple attributes into a set:

```
SELECT Auth.*
FROM Auth, (SELECT DISTINCT combine(fork1, fork2, ... fork8)
            FROM Data WHERE cond(A,B1, ..Bm) ) as F
WHERE F.fork = A.fork
```

5.4.3 Optimizing range queries

The evaluation of range queries can be substantially improved by expressing rangeAncestors(x, y) as the disjoint union of three sets: leftAncestors, Inner, and rightAncestors. Inner(x, y) consists of all fork nodes inside $(x, y]$. leftAncestors(x, y) consists of all fork nodes not in Inner whose span upper boundary intersects $(x, y]$. Likewise, rightAncestors(x, y) contains all nodes not in Inner whose span lower boundary intersects $(x, y]$. The range query $Q2$ is then equivalent

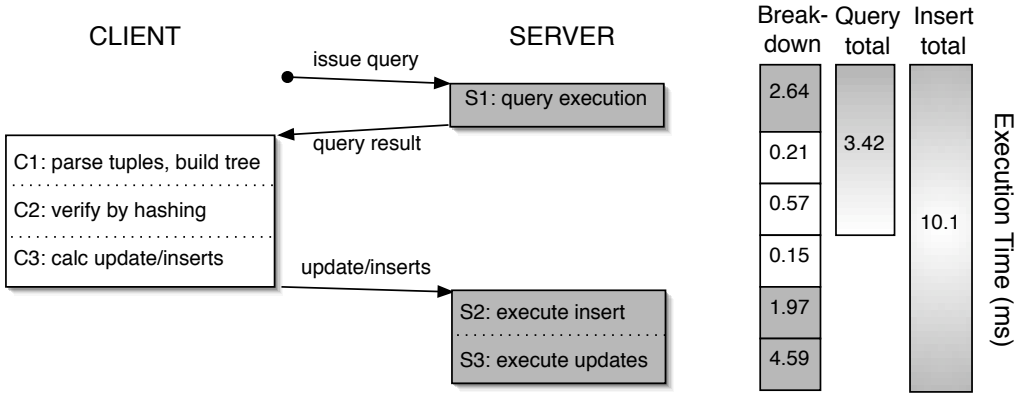

Figure 5.7: Processing diagram for authenticated QUERY and INSERT. C_1, C_2, C_3 are client-side computations, S_1, S_2, S_3 are server-side computations (including communication cost), with execution times broken down by component cost (for bundling = 4).

to the following union of three subqueries:

```
SELECT * FROM leftAncestors(x,y) as L, Auth
WHERE L.fork = Auth.fork
UNION ALL
SELECT * FROM rightAncestors(x,y) as R, Auth
WHERE R.fork = Auth.fork
UNION ALL
SELECT * FROM Auth
WHERE Auth.fork BETWEEN x AND y
```

The benefit is that the range query in the third SELECT can be evaluated using the clustered index on fork. This optimization is used in [85].

5.5 Performance evaluation

In this section we present a thorough performance evaluation of the relational hash tree and our proposed optimizations. The client authentication code was written in Java, using JDBC

to connect to the database server. Experiments were performed using both PostgreSQL, and Microsoft SQL Server databases. (Moving between database systems was very easy; the only challenge was adapting to different SQL dialects.) No substantial differences were found between systems, and for each experiment we present numbers for only one system. Both the client and server machines were Pentium 4, 2.8Ghz machines with 2 GB memory, although actual memory utilization by the client process was small. We used SHA-1 [58] as our hash function. As mentioned in Section 2.4, SHA-1 is no longer considered secure. It was broken after these experiments were performed, but moving to SHA-224 is expected to have a negligible impact on the numbers presented here. The performance numbers below do not include the invariant cost of signing the root hash upon update (4.2 ms), and verifying the root hash for queries (0.2 ms). The numbers below represent the average execution time for 200 random query or insert operations (with the 5 lowest and highest values omitted) over a database of random values.

5.5.1 Overview of cost for selection query and insert

Our experiments were run on synthetic datasets containing 200-byte tuples. On a database of 1,000,000 tuples, without authentication, a selection query on an indexed attribute takes approximately 0.6ms while an insert takes about 1.0ms. These are the baseline values used for comparison in our analysis.

Figure 5.7 shows the processing protocol for a simple selection query and an insert, along with times for our best-performing method. An authenticated query consists of execution of the hash path query at the server (quantity S_1) followed by client-side parsing the result tuples and re-building tree structure (C_1) and verification by hashing up the tree and comparing the root hash (C_2). The total time is 3.42ms, of which 77% is server computation and 23% is client time. This verified query therefore executes about 6 times slower than the baseline. Communication costs are included in the server-side costs for simplicity.

Execution of an authenticated insert includes the costs of the query *plus* the additional client computation of the updates and inserts to the relational hash tree (quantity C_3) and server execution of updates and inserts (S_2 and S_3). Overall, authenticated inserts run about 10 times slower than the baseline. The dominant cost is the execution of updates.

Our bundling optimization targets this cost, bringing it down from $14ms$ to the value in Fig. 5.7 of 4.59ms (for bundle height 4). The cost is significant because for a relational hash tree of average height h, approximately h tuples in the Auth table must be updated since the hash values of the nodes have changed. This cost is targeted by our bundling optimization described next.

5.5.2 Impact of domain tree bundling

Recall that our bundling optimization was defined in terms of a bundle height parameter b, the tree height of bundles, which is 1 for no bundling. Fig. 5.8 shows the impact of bundling on authenticated selection queries and inserts. Each bar in the graph also indicates the breakdown between client and server computation. Bundling primarily speeds up server operations by reducing the number of tuples retrieved and/or updated. The impact of bundling is dramatic for inserts, where the time for $b = 4$ is about half the time for $b = 1$. This is a consequence of fewer updates to the bundled value tree nodes in the Auth table (5 instead of about 16 without bundling).

5.5.3 Range and Arbitrary queries

The bundle height $b = 4$ is optimal not only for inserts (shown above) but for range and arbitrary queries studied next, and all results below use the bundling technique. Range queries (using the optimization described in Sec. 5.4.3) run very efficiently, just 2-3 times slower than the baseline case. Arbitrary queries are slower because disparate parts of the tree must be retrieved, and each node retrieved requires and index probe. They run about 20-30 times slower than the baseline, and scale roughly linearly with the result size. Inlining is a critical optimization, as shown in Fig. 5.9 (left) which improved processing time by about a factor of 5 in our experiments. The figure shows the per-tuple speed-up for processing an arbitrary query. That is, a value of 1 indicates that an arbitrary query returning 1000 tuples takes the same time as 1000 separate selection queries returning 1 tuple.

Scalability Fig. 5.9 (right) shows that our techniques scale nicely as the size of the database grows. The table presents selection queries, range queries, and inserts for databases consisting of 10^5, 10^6, and 10^7 tuples. Authenticated operations are roughly constant as

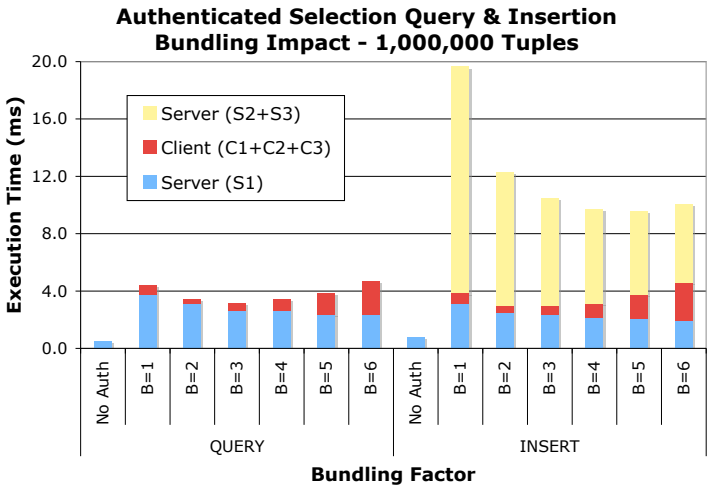

Figure 5.8: Impact of bundling on execution times for authenticated query and insert.

the database size grows. This is to be expected since at the client the execution time is determined by the result size, and at the server the execution time is largely determined by the length of paths in the relational hash tree which grow logarithmically with the database size.

5.5.4 Impact of domain size and data distribution

Recall that parameter k of the domain tree is determined by the number of bits required to represent elements in the domain of the sort attribute A, and was set to 32 in most of our experiments. Since k is the height of the domain tree, k bounds the length of paths in the value tree, and determines the size of the sets returned by Ancestors and rangeAncestors. This means that as k increases, the number of index probes increases with the average depth of nodes in the domain tree. This increases the server query time, but is mitigated by the bundling optimization. Further, the more important factor is the length of paths in the value tree because this determines the number of nodes returned by the server, as

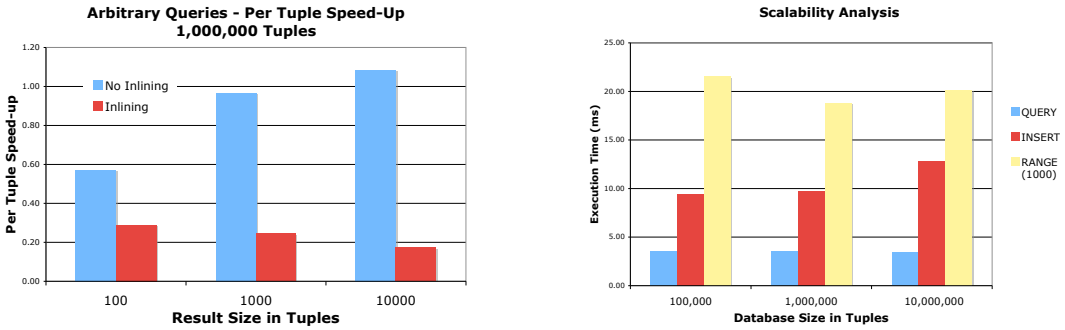

Figure 5.9: (Left) Per-tuple speed up for arbitrary condition queries using inlining. (Right) Scalability graph showing execution time for selection queries, inserts, range queries for databases with size $10^5, 10^6$ and 10^7.

well as the number of updates/inserts to the database when hashes are recomputed upon update. The length of paths in the value tree is determined by the database instance size and distribution of values across the sort attribute. Prop. 5.3.7 proves an upper bound on the height for two cases. Our experiments confirmed that the value tree is well-balanced and these paths are short. For example, there is no measurable difference in query execution times between a uniform distribution, a database of consecutive values, or distributions derived from compound sort keys. Thus the shape of the interval tree is a very effective alternative to explicit balancing of the hash tree stored at the server.

5.5.5 Storage and Communication overhead

Communication cost is included with the server-side query execution costs in all performance numbers above, and was not substantially increased by our techniques. Storage overhead was also modest: total size on disk (including indexes) for the authenticated database was 321MB compared to 249MB for the baseline.

Overall, we consider the costs presented modest, especially since the overhead of our techniques is measured by comparison to some of the fastest operations that can be performed

in a database system (index lookups and updates). In a real application, the overhead of integrity could easily be dwarfed by a query that included even a simple table scan.

5.6 Multiple party integrity

Conventional notions of authenticity typically refer to a single author. There are therefore some basic underlying challenges to formalizing authenticity of data modified by many parties. A straightforward extension of the single party case we have studied so far permits multiple authors, who all trust one another, but do not trust the database server. In this case any author is permitted to update the root hash, and the authors use a shared signature to prevent the server from modifying the root hash. Such a scenario presents some challenges for concurrent processing of updates because there is contention for the root hash value.

Nevertheless, a more realistic model for multiple party integrity has n mutually untrusting authors contributing to a common database which is stored at an untrusted server. In this setting we partition the tuples of the database by author, and ownership of a tuple is indicated by the presence of an author field in our modified schema: R'(author, A, B1 ... Bm). A query over R can be evaluated over R' and the same integrity properties of correctness, consistency, and completeness are relevant in this setting. However, it should be impossible for author α_1 to add, modify, or delete tuples in the name of any other author. Our techniques can be used to provide these integrity guarantees by prepending the author field to whichever sort key is used for the authentication table Auth, and essentially maintaining separate hash trees for each author. This relieves contention for the root hash, and improves concurrency. This model can be extended to support transfer of ownership amongst users, but we leave this as future work.

5.7 Related work

Integrity of published data

The W3C XML Signature format has been used to implement "content extraction signatures" [24], designed to allow an author to sign a document along with a definition of certain permissible operations for deriving new data. An authorized recipient can blind or extract the document and generate a signature without contacting the author. However, verification

of the signature by a third-party requires contacting the original author who will check the extractions were legal and verify the new signature. This execution model differs from ours. We would like to avoid contact with the original source of the data which restricts flexible data exchange. Bertino et al. [16] propose a framework of cooperative updates to a document which are controlled according to confidentiality and integrity processes. A drawback of their scheme is that the flow of the document through a sequence of collaborating parties must be predetermined by the first author, again restricting flexible data exchange.

Data provenance and annotation management

A number of projects have studied data provenance, or lineage, which involves tracing and recording the origin of data and its movement among databases. Computation of the set of tuples in a database that contribute to a particular tuple in the output of a query over that database is studied in [33]. In [25] the authors refer to this problem as why-provenance (which reports on why a tuple is in the answer to a query) and distinguish it from where-provenance (which reports on the origin of the data). Where-provenance is closely related to propagating source annotations through views. In [26] the authors study backward propagation: given a tuple in the output of a view to be annotated, determine the set of tuples should be annotated in the base relations to produce the annotation (with minimal side-effects). In [129] a definition for query containment is studied for queries that propagate annotations, and a prototype system for annotation management was presented recently [28]. These results may provide useful tools for supporting our integrity annotations. However, our goal is not just to carry annotations, but to provide cryptographic evidence of source attribution.

The authors of [132] describe an information integration system where query results are tagged with references to the sources from which they are derived. Similar techniques are applied to semistructured data in [89]. A general discussion of data annotation as "superimposed" information is provided in [91]. Annotea [131] is a W3C open project oriented toward tools for creating and managing web annotations, which are stored on specified servers for public use.

Hash trees and authenticated data structures

Hash trees were developed by Merkle and used for efficient authentication of a public file [97, 95] as well as a digital signature construction in [96]. Merkle's hash tree can be described as an authenticated dictionary data structure, allowing efficient proofs (relative to the root hash) of membership or non-membership of elements in the set. Authenticated dictionaries were adapted and enhanced to manage certificate revocation lists in [83, 107].

Authenticated dictionaries were adapted to relations in [47, 46], where algorithms based on Merkle trees and refinements in [107] are proposed for authenticating relations and verifying basic relational queries. In [111] the authors envision authenticated B-trees and the application of commutative hash functions to improve certain performance metrics, but abandon guarantees of completeness for queries, and provide no implementation. To our knowledge, no implementation and thorough performance evaluation of B-tree based techniques has been performed.

Our work draws on the above results (providing the integrity guarantees of [47]) but offers the first design and implementation of an authenticated database using a database management system. The algorithms in existing work were described with a main memory implementation in mind: the data structures are tree-based, and the algorithms require pointer traversals and modifications that are not appropriate for a relational DBMS. Further, the algorithms are evaluated merely in terms of worst case computational and communication complexity, which can hide critical factors in the implementation. For example, we show that minimizing the number of hashes is not a major concern, but a tree organization that minimizes index lookups is critical.

Authenticating queries using the techniques above may require revealing some data items that are not in the query answer. It is the goal of [98, 110] to provide authenticated answers while also maintaining certain secrecy properties. We focus exclusively on integrity in this chapter. Finally, in [124], hashing is used in conjunction with a trusted external party to make audit logs tamper resistant.

Chapter 6

CONCLUSION

This dissertation has addressed the challenges of providing confidentiality and integrity guarantees for complex data exchanged across heterogeneous systems. The subtle disclosures that may result, even under the correct enforcement of access control policies, pose a risk in all settings, but are particularly dangerous for distributed data exchange where it is more difficult to prevent collusion and moderate the behavior of authorized users. In addition, since heterogeneous distributed systems are not trusted, cryptographic techniques must be employed to ensure security properties. The contributions of this dissertation can help data owners and participants achieve the promise of convenient access to data and flexible collaboration, while avoiding the peril of data misuse.

6.1 Review of contributions

In Chapter 3 a novel analysis of disclosure was presented, inspired by the perfect secrecy [122] standard applied by Shannon to cryptosystems. The analysis captures subtle partial disclosures that may occur when a data owner publishes one or more views, and seeks to protect the answer to a query. We show that for simple relational queries and views, it is possible to decide query-view security, and we provide tight complexity bounds for the decision problem. The definition of query-view security has some surprising consequences. For example, it shows that simply removing sensitive columns from a relational table—a common strategy in practice—does not totally protect the sensitive data items. The analysis can account for some forms of knowledge the user may already have, resulting in a notion of query-view security relative to prior knowledge. It can also address the disclosure resulting from collusion by authorized parties.

In Chapter 4 we described a framework for protected data publishing, which uses encryption to enforce access control on data that is stored at untrusted hosts. The framework allows the data owner to go from high-level access control policies to physically-encrypted

documents in an automatic way. The resulting encrypted data consists of a single database instance that can be published to all subjects. Sets of cryptographic keys are used to negotiate access by authorized users.

The framework includes an intermediate representation (the tree protection from Section 4.3), which models the protected database with a precise access semantics. This abstract model is important because it can be used as the basis of logical manipulations to optimize the security or processing efficiency of the encrypted instances. The abstract model of protection also forms the basis for a formal security analysis.

In Chapter 5 we described a framework for authentic data exchange, and then described implementation techniques for a key component of that framework. These techniques allow a client of an untrusted database to efficiently store and maintain authentication metadata. When querying the untrusted database, clients retrieve the authentication metadata and use it to verify that data has not been modified. When updating the database, clients must also update the authentication metadata. We proposed a novel representation of the hash tree metadata for efficient storage and maintenance in a relational database. The performance analysis proves that the cost of integrity is modest, and that these techniques can be used to prevent tampering in data exchange scenarios that use relational systems as a storage mechanism.

6.2 Future directions

There are a number of remaining challenges for securing distributed data exchange. These include specific challenges that follow from the contributions of this work, as well as the overarching problem of limited adoption of security technologies in practice.

6.2.1 Adoption and deployment of security technology

The degree of adoption of security technologies is often disappointing, with many proposed technologies never deployed beyond research prototypes. Usability issues are a major reason for poor adoption [135].

The publishing framework described in Chapter 4 has the potential to ease the burden of securing distributed data. The encryption of data is driven by access control policies,

so that a user or administrator does not have to implement and apply complex encryption algorithms. But in some cases security goals and usability goals are directly in conflict. For example, in Chapter 5 our goal was to distribute data with verifiable proof of authorship and authenticity. Authenticity guarantees restrict the modification of data by unauthorized parties. They also restrict the reuse of data because derived versions or extracted pieces of the data are not authorized. There is a tension therefore between the security goal of resisting modification, and the usability objective of allowing reuse, derivation, and extraction. Negotiating these competing concerns is a compelling direction for future work.

A related challenge for practical database security is that security features are often implemented across both the database server and the application code built on top of the database. Developers often use some basic security features of the underlying database system, but implement customized security features outside the database. This may be largely the result of a failure in database system security to provide adequate tools for application developers. For example, one well-known limitation of database access control is that fine-grained, tuple-level access is not possible or practical in many systems. There are many reasons to minimize the implementation of security in applications as opposed to the database. First, if each application needs to re-implement common features, errors are more likely. Second, it is harder to verify or reason about overall system properties if security features are in hybrid form. Thus a direction for future investigation is how to provide the secure data management capabilities application developers can use to avoid re-implementation.

6.2.2 Disclosure

Many databases are dynamic, that is, they change over time. When a dynamic database is protected, the disclosures that may result from the adversary witnessing frequent changes must be taken into account. Note that this is not a problem for the definition of security presented in Chapter 3. When a view reveals nothing about a query, then subsequent versions of that view cannot reveal anything about the query either. For weaker notions of security [36], as well as for other statistical protection techniques, and for anonymization [127], this is not the case. Capturing subtle disclosures in a dynamic setting remains an

interesting area of inquiry.

In Section 2, we described a range of protection mechanisms including partial or complete encryption, protection by publishing only relational views or aggregates, anonymization, etc. In real applications, these mechanisms may be applied in combination, or an adversary may combine data from the same source that has been protected in differed ways. There is currently no unified framework for the analysis of disclosure that may result from these different protection mechanisms.

6.2.3 Confidentiality for distributed data

Dynamic data is also the context for open problems related to encrypted publishing. When the underlying database changes, policies must be re-evaluated, the encrypted database re-published, and new keys possibly disseminated to users. The techniques described in Chapter 4 can easy support these scenarios, but may be inefficient because the entire dataset must be transmitted even for small changes to the database. A preferred strategy would generate incremental updates to the encrypted document. There are connections here to the techniques of incremental view maintenance [75]: the policy queries are the views that must be maintained under updates to the underlying data. But there are also novel issues here because of the added complexity of tree protections and the normalization and rewritings applied to them to generate the encrypted XML. Changes to access control policies or user roles, even in the absence of changes to the data, also require updates to the published data.

6.2.4 Integrity for distributed data

Many open issues remain in the realization of authentic data publishing. Some of these issues have been addressed by the author in work not included in this dissertation. In [105] we investigate some of the challenges and trade-offs of signature granularity to protect data authenticity. We also propose the use of homomorphic signatures in an attempt to efficiently provide authenticity while at the same time permitting controlled derivation from signed data. A general infrastructure for storing, propagating, and processing integrity annotations is needed. Recent research by others [16] has addressed this problem (as described in Section 5.7). Finally, the authenticity of data authored by multiple parties requires further study,

as described in Section 5.6.

Security features are often treated as an afterthought. They are integrated into existing systems at great expense to mitigate the risk of inappropriate use. Instead, secure data management should be seen as an *enabling* technology. When data management tasks are trusted, new forms of communication, interaction and collaboration become possible since participants are comfortable releasing data or are certain of its authenticity. Hopefully, the contributions of this dissertation, along with the work of others in the research community, will allow the development of novel applications.

BIBLIOGRAPHY

[1] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *IFIP International Conference on Theoretical Computer Science*, Sendai, Japan, 2000.

[2] Martín Abadi and Bogdan Warinschi. Security analysis of cryptographically controlled access to xml documents. In *Principles of Database Systems (PODS)*, 2005.

[3] Nabil R. Adam and John C. Wortmann. Security-control methods for statistical databases. *ACM Computing Surveys*, 21(4):515–556, Dec. 1989.

[4] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep a secret: A distributed architecture for secure database services. In *Conference on Innovative Data Systems Research (CIDR)*, pages 186–199, 2005.

[5] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *SIGMOD Conference*, pages 86–97, 2003.

[6] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.

[7] Kazumaro Aoki and Helger Lipmaa. Fast Implementations of AES Candidates. In *The 3rd Advanced Encryption Standard Candidate Conference*, pages 106–120. NIST, 13–14 2000.

[8] Berkeley db xml. Available at `www.sleepycat.com`.

[9] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In *Conference on Computer and Communications Security (CCS)*, pages 220–230, New York, NY, USA, 2003. ACM Press.

[10] François Bancilhon and Nicolas Spyratos. Protection of information in relational data bases. In *Conference on Very Large Databases (VLDB)*, pages 494–500, 1977.

[11] François Bancilhon and Nicolas Spyratos. Algebraic versus probabilistic independence in data bases. In *Principles of Database Systems (PODS)*, pages 149–153, 1985.

[12] Josh Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In *CRYPTO*, pages 27–35, 1988.

[13] E. Bertino, S. Castano, and E. Ferrari. Securing XML documents with Author-X. *IEEE Internet Computing*, May/June 2001.

[14] Elisa Bertino, Barbara Carminati, and Elena Ferrari. A temporal key management scheme for secure broadcasting of xml documents. In *Conference on Computer and Communications Security (CCS)*, pages 31–40, New York, NY, USA, 2002. ACM Press.

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

[16] Elisa Bertino, Giovanni Mella, Gianluca Correndo, and Elena Ferrari. An infrastructure for managing secure update operations on xml data. In *Symposium on Access control models and technologies*, pages 110–122. ACM Press, 2003.

[17] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[18] Scott Boag, Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, May 2003.

[19] Philip Bohannon, Markus Jakobsson, and Sukamol Srikwan. Cryptographic approaches to privacy in forensic DNA, databases. In *Public Key Cryptography*, 2000.

[20] Luc Bouganim, François Dang Ngoc, and Philippe Pucheral. Client-based access control management for xml documents. In *Conference on Very Large Databases (VLDB)*, pages 84–95, 2004.

[21] Luc Bouganim, François Dang Ngoc, Philippe Pucheral, and Lilan Wu. Chip-secured data access: Reconciling access rights with data encryption. In *Conference on Very Large Databases (VLDB)*, pages 1133–1136, 2003.

[22] Luc Bouganim and Philippe Pucheral. Chip-secured data access: Confidential data on untrusted servers. In *Conference on Very Large Databases (VLDB)*, pages 131–142, 2002.

[23] Bouncy Castle. Open implementation of java cryptography api. *www.bouncycastle.org*.

[24] Laurence Bull, Peter Stanski, and David McG. Squire. Content extraction signatures using xml digital signatures and custom transforms on-demand. In *Conference on World Wide Web*, pages 170–177. ACM Press, 2003.

[25] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[26] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *PODS '02*, pages 150–158, 2002.

[27] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[28] Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *SIGMOD Conference*, pages 942–944, 2005.

[29] SungRan Cho, Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Divesh Srivastava. Optimizing the secure evaluation of twig queries. In *Conference on Very Large Databases (VLDB)*, pages 490–501, 2002.

[30] James Clark. XML path language (XPath), 1999. Available from the W3C, `http://www.w3.org/TR/xpath`.

[31] Privacy Rights Clearinghouse. Chronology of data breaches reported since the choicepoint incident. available at http://www.privacyrights.org/, 2005.

[32] Jason Crampton. Applying hierarchical and role-based access control to xml documents. In *ACM Workshop on Secure Web Services*, pages 41–50, 2004.

[33] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *International Conference on Data Engineering*, pages 367–378, 2000.

[34] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *CARDIS*, pages 277–284, 1998.

[35] T. Dalenius. Towards a methodology for statistical disclosure control. *Statistik Tidskrift*, 15:429–444, 1977.

[36] Nilesh Dalvi, Gerome Miklau, and Dan Suciu. Asymptotic conditional probabilities for conjunctive queries. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2005.

[37] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for xml documents. *Transactions on Information and System Security (TISSEC)*, 2002.

[38] Judith DeCew. Privacy. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Available at: http://plato.stanford.edu /archives/sum2002/entries/privacy/, Summer 2002.

[39] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Co., 1982.

[40] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[41] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[42] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with stored. In *Conference on Management of data (SIGMOD)*, pages 431–442, New York, NY, USA, 1999. ACM Press.

[43] Alin Deutsch and Yannis Papakonstantinou. Privacy in database publishing. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2005.

[44] Alin Deutsch and Val Tannen. Containment and integrity constraints for xpath. In *Knowledge Representation meets Databases Workshop (KRDB)*, 2001.

[45] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of xml documents. In *Conference on Computer and Communications Security (CCS)*, pages 136–145. ACM Press, 2001.

[46] Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314, 2003.

[47] Premkumar T. Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *IFIP Work. on Database Security*, 2000.

[48] W. Diffie and M.E. Hellman. New directions in cryptography. In *IEEE Transactions on Information Theory*, pages 644–654, 1976.

[49] Josep Domingo-Ferrer. A provably secure additive and multiplicative privacy homomorphism. In *Information Security Conference, ISC*, volume 2433 of *Lecture Notes in Computer Science*, pages 471–483. Springer, 2002.

[50] Xin Dong, Alon Y. Halevy, and Igor Tatarinov. Containment of nested xml queries. In *Conference on Very Large Databases (VLDB)*, pages 132–143, 2004.

[51] Donald Eastlake and Joseph Reagle. Xml encryption syntax and processing. http://www.w3.org/TR/xmlenc-core, 3 October 2002. W3C Proposed Recommendation.

[52] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical report, Technical University of Graz, Austria, 1980.

[53] Charles Elkan. Independence of logic database queries and update. In *Principles of database systems (PODS)*, pages 154–160, 1990.

[54] Alexandre Evfimievski, Johannes Gehrke, and Ramakrishnan Srikant. Limiting privacy breaches in privacy preserving data mining. In *Principles of database systems (PODS)*, pages 211–222, New York, NY, USA, 2003.

[55] Alexandre Evfimievski, Johannes Gehrke, and Ramakrishnan Srikant. Limiting privacy breaches in privacy preserving data mining. In *Principles of database systems (PODS)*, pages 211–222. ACM Press, 2003.

[56] Data encryption standard (DES). Federal Information Processing Standard Publication 46, 1977.

[57] Digital signature standard. Federal Information Processing Standard Publication 186-2, 2000.

[58] Secure hash standard (SHA). Federal Information Processing Standard Publication 180-2, 2000.

[59] Advanced encryption standard (AES). Federal Information Processing Standard Publication 197, 2001.

[60] R. Fagin. Probabilities on finite models. *Notices of the Am. Math. Soc.*, October:A714, 1972.

[61] R. Fagin. Probabilities on finite models. *Journal of Symbolic Logic*, 41(1), 1976.

[62] Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. Secure xml querying with security views. In *SIGMOD International Conference on Management of Data*, pages 587–598, New York, NY, USA, 2004. ACM Press.

[63] Joan Feigenbaum, Eric Grosse, and James A. Reeds. Cryptographic protection of membership lists. *Newsletter of the Intern Assoc for Cryptologic Research*, 9(1):16–20, 1992.

[64] Joan Feigenbaum, Mark Y. Liberman, and Rebecca N. Wright. Cryptographic protection of databases and software. In *Distributed Computing and Crypto*, pages 161–172, 1991.

[65] Mary 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 (TODS)*, 27(4):438–493, 2002.

[66] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native xml base management system. *The VLDB Journal*, 11(4):292–314, 2002.

[67] C.M. Fortuin, P.W. Kasteleyn, and J. Ginibre. Correlation inequalities on some partially ordered sets. *Communications in Mathematical Physics*, 22:89–103, 1971.

[68] Irini Fundulaki and Maarten Marx. Specifying access control policies for xml documents with xpath. In *ACM Symposium on Access control models and technologies (SACMAT)*, pages 61–69, New York, NY, USA, 2004. ACM Press.

[69] Alban Gabillon and Emmanuel Bruno. Regulating access to xml documents. *Proc. Working Conference on Database and Application Security*, July 2001.

[70] Ruth Gavison. Privacy and the limits of law. *Yale Law Journal*, 89:421–471, 1980.

[71] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, 2001.

[72] David K. Gifford. Cryptographic sealing for information secrecy and authentication. *Communications of the ACM*, 25(4):274–286, 1982.

[73] G. Grätzer. *General Lattice Theory*. Birkhäuser Verlag, Basel, 1978.

[74] Patricia P. Griffiths and Bradford W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3):242–255, 1976.

[75] Ashish Gupta and Iderpal Singh Mumick, editors. *Materialized views: techniques, implementations, and applications*. MIT Press, Cambridge, MA, USA, 1999.

[76] Ashish Gupta, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint checking with partial information. In *Principles of database systems (PODS)*, pages 45–55, 1994.

[77] U.s. health insurance portability and accountability act (hipaa). Available at: http://www.hhs.gov/ocr/hipaa/.

[78] Hakan Hacigumus, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database service provider model. In *SIGMOD Conference*, 2002.

[79] Hakan Hacigumus, Balakrishna R. Iyer, and Sharad Mehrotra. Providing database as a service. In *International Conference on Data Engineering (ICDE)*, 2002.

[80] Alon Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[81] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *The VLDB Journal*, 11(4):274–291, 2002.

[82] J. Killian. Efficiently committing to databases. Technical report, NEC Research Institute, February 1998.

[83] Paul C. Kocher. On certificate revocation and validation. In *Fin. Cryptography*, pages 172–177, 1998.

[84] D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *Conference on Artificial Intelligence*, pages 580–587, 1998.

[85] Hans-Peter Kriegel, Marco Potke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *VLDB Conference*, pages 407–418, 2000.

[86] M. Kudo and S. Hada. Xml document security based on provisional authorization. *Computer and Communication Security (CCS)*, November 2000.

[87] Sun Microsystems Laboratory. Xacml implementation. http://sunxacml.sourceforge.net.

[88] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *IEEE Symposium on Security and Privacy*, pages 71–85, 2004.

[89] T. Lee, Stephane Bressan, and Stuart E. Madnick. Source attribution for querying against semi-structured documents. In *Workshop on Web Information and Data Management*, 1998.

[90] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In *Conference on Very Large Data Bases (VLDB)*, pages 171–181, 1993.

[91] David Maier and Lois M. L. Delcambre. Superimposed information for the internet. In *WebDB (Informal Proceedings)*, pages 1–9, 1999.

[92] Kevin McCurley. As quoted in [6], page 1., August 1998.

[93] Catherine Meadows. Formal verification of cryptographic protocols: A survey. In *ASIACRYPT '94: Conference on the Theory and Applications of Cryptology*, pages 135–150, London, UK, 1995. Springer-Verlag.

[94] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[95] Ralph C. Merkle. Protocols for public key cryptosystems. In *Symposium on Security and Privacy*, 1980.

[96] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.

[97] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Information Systems Laboratory, Stanford University, 1979.

[98] Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *Symposium on Foundations of Computer Science (FOCS)*, 2003.

[99] Daniele Micciancio and Bogdan Warinschi. Completeness theorems for the abadi-rogaway logic of encrypted expressions. *Journal of Computer Security*, 12(1):99–129, 2004. Preliminary version in WITS 2002.

[100] Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of cryptography conference (TCC) 2004*, volume 2951, pages 133–151, Feb 2004.

[101] Gerome Miklau and Dan Suciu. Cryptographically enforced conditional access for XML. Fifth International Workshop on the Web and Databases (WebDB 2002), June 2002.

[102] Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *Conference on Very Large Databases (VLDB)*, pages 898–909, September 2003.

[103] Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, September 2003.

[104] Gerome Miklau and Dan Suciu. A formal analysis of information disclosure in data exchange. In *Proceedings of the 2004 Conference on Management of Data (SIGMOD)*, pages 575–586. ACM Press, 2004.

[105] Gerome Miklau and Dan Suciu. Managing integrity for data exchanged on the web. In *WebDB*, pages 13–18, 2005.

[106] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. Xml access control using static analysis. In *ACM conference on Computer and communications security (CCS)*, pages 73–84, New York, NY, USA, 2003. ACM Press.

[107] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *USENIX Security Symposium*, 1998.

[108] OASIS. eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml.

[109] Manuel Oriol and Michael Hicks. Tagged sets: a secure and transparent coordination medium. In *Conference on Coordination Models and Languages (COORDINATION)*, volume 3454 of *LNCS*, pages 252–267. Springer-Verlag, April 2005.

[110] Rafail Ostrovsky, Charles Rackoff, and Adam Smith. Efficient consistency proofs on a committed database. MIT LCS Technical Report TR-887, Feb 2003.

[111] HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In *International Conference on Data Engineering (ICDE)*, 2004.

[112] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, NY, 1985.

[113] James Randall. Hash function update due to potential weakness found in sha-1. RSA Laboratories, Technical Note, March 2005.

[114] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–179. Academic Press, 1978.

[115] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[116] R.L. Rivest. The MD5 message digest algorithm. RFC 1320, April 1992.

[117] Michael Rys, Don Chamberlin, and Daniela Florescu. Xml and relational database management systems: the inside story. In *Conference on Management of data (SIGMOD)*, pages 945–947, New York, NY, USA, 2005. ACM Press.

[118] Bruce Schneier. *Applied Cryptography, Second Edition*. John Wiley and Sons, Inc., 1996.

[119] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[120] 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. In *Conference on Very Large Data Bases (VLDB)*, pages 65–76, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[121] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Conference on Very Large Data Bases*, pages 302–314. Morgan Kaufmann, 1999.

[122] C. E. Shannon. Communication theory of secrecy systems. In *Bell System Technical Journal*, 1949.

[123] Claude E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27(3):379–423, July 1948.

[124] Richard T. Snodgrass, Shilong Stanley Yao, and Christian Collberg. Tamper detection in audit logs. In *VLDB Conference*, 2004.

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

[126] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 2nd edition, 2002.

[127] Latanya Sweeney. k-Anonymity: a model for protecting privacy. *Int. J. on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5), 2002.

[128] Tamino xml server. Available at `www.softwareag.com`.

[129] Wang-Chiew Tan. Equivalence among relational queries with annotations. In *International Workshop on Data Base and Programming Languages (DBPL)*, 2003.

[130] Auguste Kerckhoffs (von Nieuwenhof). La cryptographie militaire. (French) [Military cryptography]. *Journal des Sciences Militaires*, IX(1), January 1883.

[131] W3C. Annotea project. http://www.w3.org/2001/Annotea/.

[132] Y. Richard Wang and Stuart E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB*, pages 519–538, 1990.

[133] S. Warren and L. Brandeis. The right to privacy. *Harvard Law Review*, 4:193–220, 1890.

[134] Alan F. Westin. *Privacy and Freedom*. Atheneum, New York, 1967.

[135] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *8th USENIX Security Symposium*, 1999.

[136] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.

[137] Xyleme server. Available at `www.xyleme.com`.

[138] XSL Transformations (XSLT), version 1.0. http://www.w3.org/TR/xslt, 16 November 1999. W3C recommendation.

[139] Xiaochun Yang and Chen Li. Secure xml publishing without information leakage in the presence of data inference. In *Conference on Very Large Databases*, pages 96–107, 2004.

[140] A. Yao. How to generate and exchange secrets. In *Symposium on Foundations of Computer Science (FOCS)*, volume 13, pages 162–167, 1986.

[141] Andrew Chi-Chih Yao. Protocols for secure computations. In *Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.

[142] Ting Yu, Divesh Srivastava, Laks V. S. Lakshmanan, and H. V. Jagadish. Compressed accessibility map: Efficient access control for xml. In *Conference on Very Large Databases (VLDB)*, pages 478–489, 2002.

[143] Ting Yu, Divesh Srivastava, Laks V. S. Lakshmanan, and H. V. Jagadish. A compressed accessibility map for xml. *ACM Transactions on Database Systems*, 29(2):363–402, 2004.

VITA

Gerome Miklau was born in Paris and raised in New York City. He received dual bachelor's degrees in Mathematics and in Rhetoric from the University of California, Berkeley in 1995. He received an M.S. in Computer Science in 2001 and a Ph.D. in Computer Science in 2005, both from the University of Washington. Beginning September 2005, he will be an Assistant Professor at the University of Massachusetts, Amherst.