Controlling Access to Published Data Using Cryptography

Gerome Miklau Dan Suciu

{gerome, suciu}@cs.washington.edu
University of Washington, Seattle, WA, USA

Abstract

We propose a framework for enforcing access control policies on published XML documents using cryptography. In this framework the owner publishes a single data instance, which is partially encrypted, and which enforces all access control policies. Our contributions include a declarative language for access policies, and the resolution of these policies into a logical "protection model" which protects an XML tree with keys. The data owner enforces an access control policy by granting keys to users. The model is quite powerful, allowing the data owner to describe complex access scenarios, and is also quite elegant, allowing logical optimizations to be described as rewriting rules. Finally, we describe cryptographic techniques for enforcing the protection model on published data, and provide a performance analysis using real datasets.

1 Introduction

There is an ever-increasing amount of data available in digital form, and almost invariably that data is now near a network. Recent research into integration systems and peer-to-peer databases has created new ways for diverse groups to share and process data [23, 34, 21, 28]. But in most practical cases, complex constraints of trust and confidentiality exist between these cooperating or competing groups. As a result, in many cases data is disseminated only when there are no security or confidentiality issues among any possible recipient. This means data that could safely be disseminated to certain parties remains hidden behind a firewall or server.

It is not just database researchers with new research tools that would like to encourage data sharing and dissemination. In the case of scientific data,

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

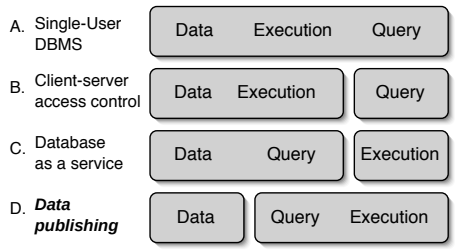


Figure 1: Organization of related problems in terms of trust domains and key objects Data, Query origination, and query Execution resources.

U.S. legislation has been passed requiring data underlying federally-funded research to be published upon request [3]. The U.S. National Institute of Health (a major funding agency for medical research) will expect all funded scientists to release their final research data for use by other scientists [29]. In addition to funding agencies, some scientific journals are making data sharing a condition of publication [27], while others are coming under pressure to do so [24].

The trust, confidentiality and security issues involved in sharing data are immense however [18, 31], and there are few tools for managing access to the data that peers are encouraged or required to share (especially when distributed data processing is needed). In this paper we introduce a framework to allow a database owner to express rich access policies, generate a single, partially-encrypted version of the data that enforces *all* access policies, publish it, and then enforce the policies by granting keys to users.

Database security has been studied extensively in the past [13, 1, 9], however none of that work applies to data publishing. Consider Figure 1, which classifies scenarios based on trust domains. The trivial case is that of single-user database applications, which have a single trust domain, Figure 1-A, and thus raises no security issues. Most of the work in database security fits in the scenario of Figure 1-B, where the data owner controls the execution en-

gine, but doesn't trust the client asking the queries. Client-server database applications, as well as many Web-based database applications fall into this case. Traditional work in security has focused on the data owner: how to respond to queries without revealing protected data. More recent work has also addressed the client's security [10]: how to preserve client privacy by not allowing the data owner to witness all their queries. A newer class of applications, *database as a service* [22], has a different partition into trust domains: Fig 1-C. A client owns the data and also issues the queries, but would like to pay an untrusted party to store the data and execute the queries on their behalf. *Data publishing*, the focus of this paper, fits under none of these scenarios – its trust domains are described in Fig 1-D. In this case, once the data owner has published the data, she loses control over it: it can be downloaded, copied, disseminated, redistributed. Both query origination and the execution engine are in a different trust domain from the data.

Contributions We propose a comprehensive framework for access control over published data, using encryption. The components of the framework are shown in Fig. 2. The data owner starts by defining high-level access control policies over its data: our *first* contribution is to define a language for specifying such policies as queries. A modified query engine evaluates these policy queries to produce a result which is not data, but a single "protection" over the XML document. Our *second* contribution consists of a logical data model for these protections. The model is simple, yet powerful enough to express complex policies, has a clean semantics, and admits some simple logical optimizations, which later result in significant space savings for the published data. Next, we need to translate this logical model into a partially encrypted XML document. Our *third* contribution consists of showing how to perform this encryption, using the recent W3C Recommendation "XML Encryption Syntax" [15] as an encryption format. We adapt and extend known techniques: secret sharing [32, 4], bounded key encryption [16, 17], and random sequences [33].

At this point in our framework, the partially encrypted XML document is published on the Internet. Notice that this is a *single* XML document, which enforces *all* access policy queries. Once published, the data owner relinquishes all control over who downloads and processes the data, so we depend entirely on encryption to enforce our control policies. A legitimate client can access the data conditionally, depending on the keys they possess. These keys will be conventional cryptographic keys (i.e. random bit sequences, obtained from the data owner) or data values (e.g. the `name`, `social-security number`, and `date-of-birth` of a patient's confidential data), or "inner" keys (random bit sequences, obtained from other parts of the data). The client does not, however, need to decrypt the entire data instance: they can access it selectively, using a query language, while supplying appropriate keys. Our *fourth* contribution is a simple extension to XQuery to support this. Finally, our *fifth* contribution consists of an experimental evaluation, testing the feasibility of such an approach. We are primarily concerned with the size of the partially encrypted document, and the encryption/decryption speed. We show that they are reasonable, and can be dramatically improved using a combination of logical and physical optimizations.

2 Motivating Example

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 our 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 laws are enforced.

Our policy queries use an extension of XQuery with a KEY and a TARGET clause. The first policy query is motivated by the relationship between primary and competing researcher:

```
Policy Query 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 reference. 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.

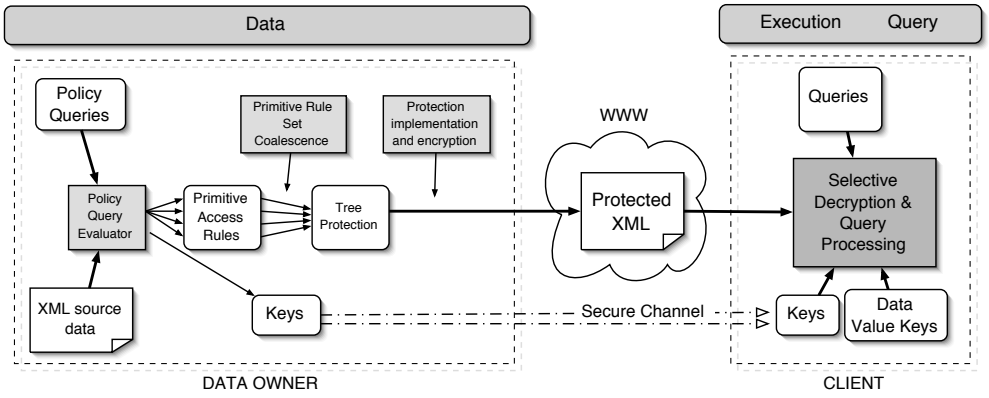


Figure 2: The protected data publishing framework.

The intent of Policy Query 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 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 who have 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 (see Fig. 2). She can thus have very fine-grained control over how users access the data.

Notice that the targets of Policy Queries 2.1 and 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 2.1 or with the key in Query 2.2.

Policy Query 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 of them has 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 2.4
```
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

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

users have access to the HIV test, hence the auditor's query is:

```
Policy Query 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 2.1) have access to the analysis/tests/HIV data.

Current Approaches to Access Control There are two traditional approaches to controlled access to data. The first keeps the data on a *secure server* that authenticates users and enforces the access policies, without publishing the data. The other is to publish *multiple views* of the data, one for each user or class of users. This is the current approach to controlled data publishing, and has several drawbacks. One is that the number of views may become very large: for example, to implement Policy Query 2.2 one needs a different view for each subject in the database. Another is that it prevents dissemination: users cannot further publish the data that they downloaded from the owner. Finally, it makes evolution harder.

Our Solution In our approach all policy queries are executed by the policy query evaluator (see Fig. 2) and then a *single* partially-encrypted XML document, enforcing all policies, is published. Users need proper keys to access restricted data. The document can be downloaded, copied, and its accessible parts partially processed by a certain party before being passed along to another party with different access rights.[2] The keys are maintained and managed by the data owner, who grants them to specific users. Key transmission may be delayed in a data commitment scenario, or interactive key protocols can be used to enforce complex access control policies.

3 The Tree Protection Logical Model

Assuming we encrypt the published data, it is far from clear how to enforce multiple, overlapping policy queries on a single data instance. We describe here a logical model for protecting an XML tree, which plays a central role in our secure 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, attributes, and children are all hidden. We present this formally next.

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 $\text{NODES}(t)$ and the set of edges with $\text{EDGES}(t)$, and $\text{VALUE}(i)$ 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. 2, Examples of exchange keys are: "registration" in Query 2.1, and the subject's keys in Query 2.2, with system generated names like "subject030223". We denote by XchgKey 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 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.[3] We denote by InnerKey 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 128-bit string by using the UTF-8 encoding, and dividing the result into 128 bit blocks which are XOR'd together. We use them 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 2.1 is a data value key. We write $\text{Key} = \text{XchgKey} \cup \text{InnerKey} \cup \text{DataValue}$.

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

[2] Of course, we cannot prevent a malicious user from decrypting the data and publishing it. But we can allow authorized users to re-publish the encrypted data after processing or transforming accessible portions.

[3] Inner keys are needed to support complex access control policies, and are used in the normalization process described later in Sec. 3.2.

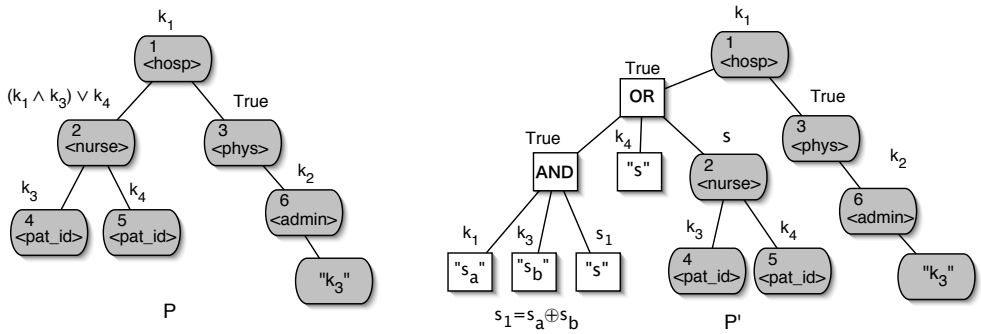


Figure 3: A tree protection P (Example 3.1), and an equivalent normalized protection P' (Example 3.4). The white nodes are metadata nodes.

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 $\text{VALUE}(i) \in \text{DATAVALUE} \cup \text{INNERKEY}$.

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 $\text{trim}(t) = t$.

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 $\varphi_i = \wedge_{j \preceq i} \sigma_j$.

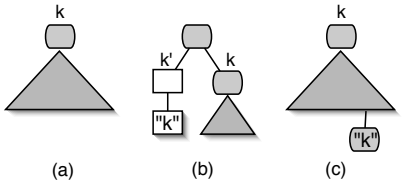


Figure 4: Typical usage patterns of tree protections, described in Example 3.2.

Example 3.1 Figure 3 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 $\sigma_2 = (k_1 \wedge k_3) \vee k_4$ and necessity formula $\varphi_6 = k_1 \wedge \text{true} \wedge k_2$. There are no metadata nodes in this case, so $t = t_m$.

Example 3.2 Figure 4 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.

3.1 Semantics

The semantics of protection P is a function, $acc_P(K)$, which, given a set of keys K, returns a set of nodes that can be accessed by the user "knowing" K. The function will return only the data nodes that the user can access: during the process described below she may also access metadata nodes, but in the semantics we only care about which data nodes she can access. The input K will be restricted to $K \subseteq \text{XCHGKEY} \cup \text{DATAVALUE}$, 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 has currently 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 increases N by adding all nodes that she can unlock by "using" the current the 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 $M' = combine(M)$ which, given a finite set of keys $M \subseteq \text{KEY}$, 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 $combine(M) = M \cup \{r \oplus r' \mid r, r' \in M \cap \text{INNERKEY}\}$. Other choices are possible, but one has to restrict *combine* to be computationally bounded, otherwise it may return the set of all random keys[4]. 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 $acc_P(K)$ formally: $acc_P(K) = N \cap \text{NODES}(t)$, where $N \subseteq \text{NODES}(t_m)$ and $M \subseteq \text{KEY}$ are the least fix point of the following two mutually recursive definitions:

$$\begin{aligned} N &= \{i \mid i \in \text{NODES}(t_m), combine(M) \models \varphi_i\} \\ M &= K \cup \{\text{VALUE}(i) \mid i \in N\} \end{aligned}$$

Finding this fixpoint can be done with a standard iterative procedure[5] which corresponds to the informal description above.

Example 3.3 For the tree protection P illustrated in Figure 3, the following are values of $acc_P(K)$ for selected subsets $K \subseteq \text{XCHGKEY}$. (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 $K \subseteq \text{XCHGKEY} \cup \text{DATAVALUE}$, $acc_P(K) = acc_{P'}(K)$. 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.

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

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

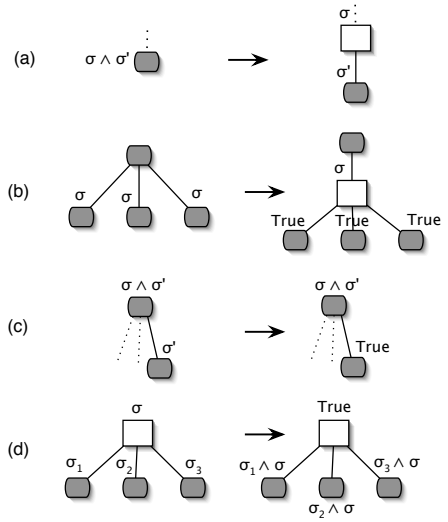


Figure 5: Protection rewritings for logical optimizations: (a) formula split; (b) formula pull-up; (c) formula simplification; (d) formula push-down.

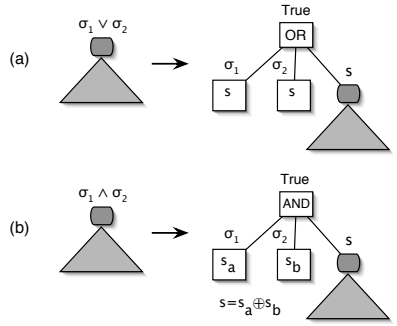


Figure 6: Protection normalization.

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 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 6 shows more rewriting rules that we use to normalize the protection before encrypting it as shown in Sec. 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:

(a) *Disjunction*: To eliminate $\sigma_1 \vee \sigma_2$ a new random key is generated, s, 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 are generated, s_a and s_b, 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 [4].

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. 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. 6.

Example 3.4 Figure 3 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 Implementing a Protection with Encryption

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 only to those nodes of t in $acc_P(K)$. The first step is to normalize P and obtain an equivalent protection P' for a metadata tree t_m. Then we encrypt P', as described next.

XML Encryption Recommendation The recent W3C Recommendation on XML Encryption Syntax and Processing [15] 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.

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 hash function 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 [16, 17] for a similar purpose. In designing it, we were inspired by the techniques of [33] for search on encrypted data.

The Encryption The actual encryption of the protection P' is done by a straightforward recursive traversal of the metadata tree of P'. Notice that after normalization, all keys in P' are atomic. 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: `EnctryptionMethod` (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.

Example 4.1 Figure 7 shows an XML instance constructed from the normalized protection P' of Figure 3. The `CipherValue` element contains bytes

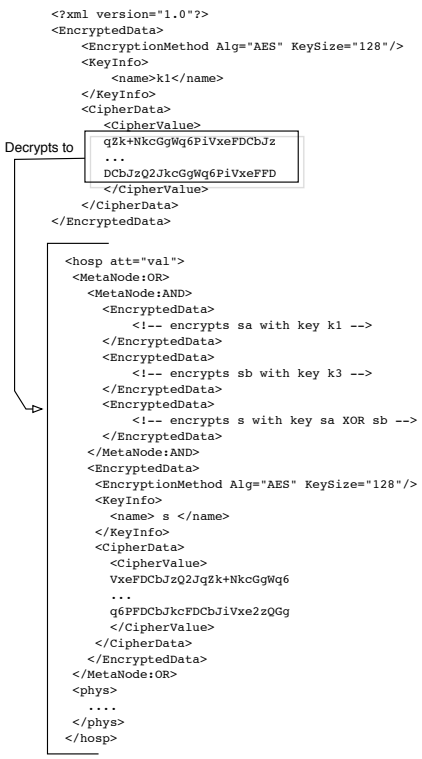


Figure 7: Encrypted XML based on protection of Figure 3.

(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. 3.2, and at the physical encryption level we can apply compression before encryption. We discuss this further in Section 7.

4.1 Security Discussion

The implementation of a protection is designed so that the following property will hold:

Property 4.2 (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 $x \notin acc_P(S)$ then the most efficient algorithm for deriving x from t' and K requires guessing missing keys.*

The first item easy to prove: 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 encryption protocol 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 [4]. Our protocol for encrypting with data value keys is also secure. However, the security of the *combined* protocols requires a separate formal proof, and this is beyond the scope of our paper.

Not captured by Property 4.2 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. They can count the number of encrypted children of a node, even if they cannot decrypt them. Some of these leakages can be avoided with improved encryption schemes, but this is beyond our scope.

5 A Policy Query Language

In this section we describe the language for writing *policy queries* and define its semantics. The language itself has already been illustrated in Sec. 2, and here we provide its complete syntax. Then we present its semantics, which is more interesting and far from obvious: a *set* of policy queries must evaluate to a *single* unified protection on the XML tree (as described in Sec. 3) and must therefore resolve possibly overlapping and contradictory policy queries.

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.

5.1 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 [6] 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.

5.2 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 some lattice-theoretic techniques [20].

Recall the definition of $acc_P(K)$ in Sec. 3.

Definition 5.1 (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. 3).

Definition 5.2 *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[6]. 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 $\sigma_1 \vee \ldots \vee \sigma_n$.

[6]Reflexive and transitive.

Theorem 5.3 *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 $i \in$ NODES(t), the key expression σ_i is given by: $\sigma_i = \bigvee\{\bigwedge S \mid \exists(S \to e) \in R_s, i \preceq e\}$ 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.*
- *$GLB(PS(R_s \cup R_n))$ 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 $S' \subseteq S$.*

The proof is included in [26].

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 $S_1 \to e_1$, $S_2 \to e_2$, ... 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 in [26] how to reduce the problem to query containment for the XQuery language. 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, [14] describes such a fragment for which the containment problem is Π_2^P-complete.

6 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. 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 6.1 Consider Policy Query 2.1 from Sec. 2. 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.

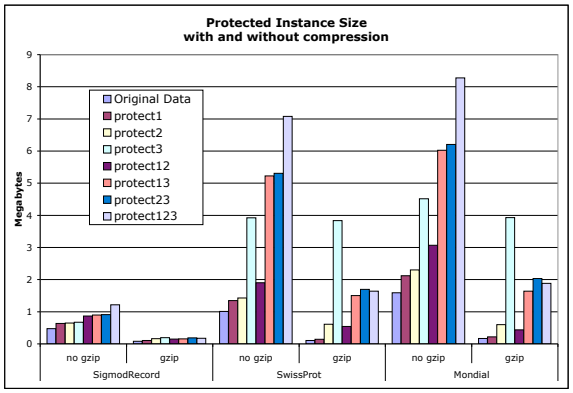
7 Performance Analysis

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 [8] of the Advanced Encryption Standard (AES) [11] with 128-bit keys. We tested other algorithms as well and while the ideal choice of algorithm is a complex issue[7] 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[8]. We consider basic protections which place

[7]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.

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



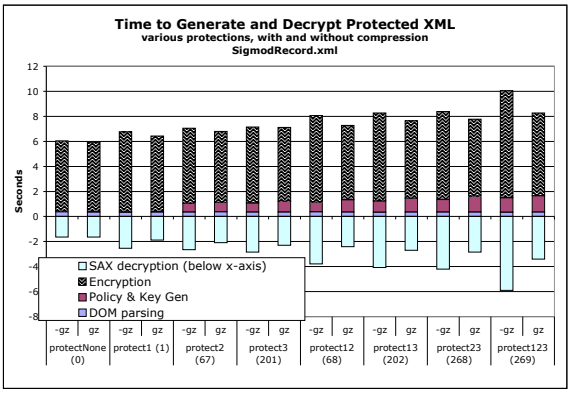


Figure 8: Size of protected documents (left), and time to generate and decrypt protected documents (right).

single unique keys at all nodes on different levels of the document trees, and are named 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 the same size as the cleartext input. 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 [15]. This results in an immediate blow-up of 33% in the case of the simplest encrypted XML. The problem is compounded during super-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 super-encryption, and also by using compression.

Figure 8 (left) 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 larger than the original, especially in the case of P_{123} which involves many keys and super-encryption: 5 times the size of original for Mondial, 7 times for SwissProt for this protection. Applying rewriting rules (Sec. 3), when possible, can help however. Protection P_{13} can be seen as an approximation of the result of applying rewriting rule (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 (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[9]. 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 on the right of Figure 8 measures the generation time (above the x-axis) and decryption time (below the x-axis)[10]. 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 [2] 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.

8 Related Work and Conclusions

A number of access control models for XML have been proposed [12, 19, 30]. These rely on a trusted processor to regulate access and do not allow secure publishing. One of these is XACML [30], a flexible standard for declaring access control policies. Our policy queries could be expressed as XACML policies. Unlike our formalism, XACML is not based on

[9]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.

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

a query language like XQuery. Rules requiring joins, like our Policy Query 2.3 (Sec. 2) are expressed in an ad-hoc syntax. The Author-X system [5] supports remote enforcement using encryption and key transmission, but this work does not describe a formal model of document protection, an expressive policy language, nor does it support data value keys. The use of data value keys to enforce binding patterns was addressed recently in the context of XML [25]. These techniques extend those developed in [7, 16, 17].

Conclusions We have provided a comprehensive framework for controlled sharing of data. Our framework includes high-level access control policies, a powerful logical model for protecting a document tree, and encryption techniques to construct an XML document that enforces the policies. We believe such a framework can satisfy the needs of emerging communities of users who want to share data in a distributed setting, but are restricted by trust and privacy constraints. Our continued work is focused on proving formal security claims about the data instances we have described here, as well as improved query processing and update techniques.

Acknowledgments Daniele Micciancio and Victor Vianu provided helpful comments. Suciu was partially supported by the NSF CAREER Grant IIS-0092955, NSF Grant IIS-0140493, a gift from Microsoft, and a Sloan Fellowship. Miklau was partially supported by NSF Grant IIS-0140493.

References

[1] N. R. Adam and J. C. Wortman. Security-control methods for statistical databases. *ACM Computing Surveys*, 21(4):515–556, Dec. 1989.

[2] K. Aoki and H. Lipmaa. Fast Implementations of AES Candidates. In *The 3rd AES Candidate Conference*, pages 106–120. NIST, 2000.

[3] B. Azar. Information act opens data and debate. *American Psychological Association Monitor*, 30(8), September 1999.

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

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

[6] S. Boag, D. Chamberlin, J. Clark, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, May 2003.

[7] P. Bohannon, M. Jakobsson, and S. Srikwan. Cryptographic approaches to privacy in forensic DNA, databases. In *Public Key Cryptography*, 2000.

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

[9] S. Castano, M. G. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley & ACM Press, 1995.

[10] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Foundations of Computer Science (FOCS)*, 1995.

[11] J. Daemen and V. Rijmen. The block cipher rijndael. In *CARDIS*, pages 277–284, 1998.

[12] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for xml documents. *ACM TISSEC*, 2002.

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

[14] A. Deutsch and V. Tannen. Containment and integrity constraints for xpath. In *KRDB*, 2001.

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

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

[17] J. Feigenbaum, M. Y. Liberman, and R. N. Wright. Cryptographic protection of databases and software. In *Distributed Computing and Crypto*, pages 161–172, 1991.

[18] The protection of human subjects' data. http://www.fmridc.org/submissions/privacy guidelines.html. fMRI Data Center, Dartmouth College.

[19] A. Gabillon and E. Bruno. Regulating access to xml documents. *Proc. Working Conference on Database and Application Security*, July 2001.

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

[21] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can peer-to-peer do for databases, and vice versa? WebDB Workshop on Databases and the Web, June 2001.

[22] H. Hacigumus, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database service provider model. In *SIGMOD Conference*, 2002.

[23] Z. G. Ives, A. Y. Levy, J. Madhavan, R. Pottinger, S. Saroiu, I. Tatarinov, S. Betzler, Q. Chen, E. Jaslikowska, J. Su, and W. Yeung. Self-organizing data sharing communities with SAGRES. In *SIGMOD '00*, page 582, 2000.

[24] S. H. Koslow. Should the neuroscience community make a paradigm shift to sharing primary data? *Nature Neuroscience*, 3(9):863–865, September 2000.

[25] G. Miklau and D. Suciu. Cryptographically enforced conditional access for XML. Fifth International Workshop on the Web and Databases (WebDB 2002), June 2002.

[26] G. Miklau and D. Suciu. Controlling access to published data using cryptography. University of Washington Technical Report (TR 03-05-03), May 2003. http://www.cs.washington.edu/homes/gerome.

[27] Nature Neuroscience Editorial. A debate over fmri data sharing. *Nature Neuroscience*, 3(9):845–846, Sep 2000.

[28] W. S. Ng, B. C. Ooi, K. L. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing. *International Conference on Data Engineering*, March 2003.

[29] NIH statement on sharing research data. *http://grants2.nih.gov/grants/policy/data_sharing/index.htm*, March 2002. U.S. National Institutes of Health.

[30] OASIS. eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml.

[31] A question of balance: Private rights and the public interest in scientific and technical databases. National Academy Press, 1999. National Research Council.

[32] A. Shamir. How to share a secret. *CACM*, 22(11):612–613, 1979.

[33] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[34] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 1996.