Distributed Access Control: A Privacy-conscious Approach

Bogdan Cautis
INRIA Futurs & U. Paris Sud
bogdan.cautis@inria.fr

ABSTRACT

With more and more information being exchanged or published on the Web or in peer-to-peer, and with the significant growth in numbers of distributed, heterogeneous data sources, issues like access control and data privacy are becoming increasingly complex and difficult to manage. Very often, when dealing with sensitive information in such settings, the specification of access control policies and their enforcement are no longer handled by the actual data sources, and are (partially) delegated to third-parties. Besides practical reasons, this is the case when decisions regarding access depend on factors which overpass the scope and knowledge of some of the entities involved. More specifically, policies may depend on *private* aspects concerning users (accessing data) or data owners. In this case, the only solution is to entrust some third-party authority with all the information needed to apply access policies. However, as the policies themselves depend on sensitive information, this outsourcing raises new privacy issues, that were not present in centralized environments. In particular, information leaks may occur during access control enforcement.

In this paper, we consider these issues and, starting from non-conventional digital signatures, we take a first step towards an implementation solution for such settings where both data and access policies are distributed. Our approach involves rewriting user queries into forms which are authorized, and we illustrate this for both structured (relational) and semi-structured (XML) data and queries.

Categories and Subject Descriptors: H.2.7 [Database administration]: Security, integrity and protection – *Access control*

General Terms: Security, Algorithms.

Keywords: access control, privacy, digital signatures, relational data, semi-structured data, XML.

SACMAT'07, June 20-22, 2007, Sophia Antipolis, France.

1. INTRODUCTION

Access control has been studied extensively, for both relational and semi-structured data, yet mostly in centralized settings. With more and more information being exchanged and published on the Web or in peer-to-peer, and with the significant growth in numbers of heterogenous, distributed data sources (such as PDAs or medical devices), issues like access control and privacy are becoming increasingly complex and difficult to manage.

Consider for instance health information [18]. By nature, a patient record consists of many pieces owned and managed by different entities (patient, referring doctor, specialists, hospitals, insurance company, various medical devices, etc). Moreover, it has often a sensitive nature, since patients clearly do not want unauthorized parties to access parts of their medical record.

Very often, for such data that is both highly distributed and partially confidential, the specification of access control policies and their enforcement are no longer handled by the actual data sources, and have to be (partially) delegated to third-parties [1, 2, 17, 10, 24]. There are several reasons in favor of such an approach. For example, access control tasks may be too resource demanding for some data devices. Also, policies may be somewhat global and not necessarily chosen by data owners, and by delegating access control to such trusted authorities, one does not risk to misinterpret policies or rely on obsolete ones.

Even more importantly, decisions regarding access may depend on various factors which could overpass the scope and knowledge of some of the entities involved. More specifically, policies may depend on *private* aspects concerning both the user (who is demanding access to resources) and the data source. In this case, the only solution is to entrust some third-party authority with all the information needed to apply access policies and let it handle their enforcement. We can imagine that such authorities are at the boundary between private domains, having access to private information from several parties and managing access policies that rely on this information.

However, when the policies themselves depend on sensitive information, this outsourcing of access control can raise new issues, mainly concerning privacy, that were not present in centralized environments. More precisely, information leaks may occur even during the initial enforcement phase, i.e., when decisions are made about how the evaluation and answering of the original user query are altered, and before actually returning query answers. Let us consider a high-level example which illustrates this situation and re-

lated ideas. (This example will be detailed further on for semi-structured data and queries.)

EXAMPLE 1.1. *Consider the 3-party scenario of Figure 1, in which a user,* `Specialized Medical Doctor (SMD)`*, is demanding access to some patient information from a medical record stored by a* `Hospital` *source.*

Assume the initial user query asks for:

"`treatment` *data for* `John Smith`*, either from the hospital's regular medical records or from* `clinical trials` *which were conducted there".*

Assume `Hospital` *does not accept queries for patients unless they are first verified, properly restricted and certified by their corresponding authority,* `Referring Medical Doctor (RMD)`*. In other words,* `RMD` *acts as an access control proxy for incoming queries from users, modifying them so that they access only allowed information. Let us assume that some access rule from* `RMD` *triggers only when*

- `SMD` *performs* `clinical trials` *(which is a piece of private information concerning* `SMD`*, not to be disclosed to parties such as* `Hospital`*),*
- *the patient has some* `blood related` *diagnosis (also a piece of private information, not to be disclosed to parties such as* `SMD`*).*

Assume that the effect of this rule will be that `SMD` *can only access the* `treatment` *data from regular medical records (but not the one from clinical trial records).*

In order to apply this rule, `RMD` *must have access to these private sensitive details from both parties. But if the rule is triggered, and the query issued by* `SMD` *is transformed accordingly, then simply disclosing to either party that the query is not the original one but was modified (and how) reveals private sensitive information.*

Notice that we assume in the above example that access control is implemented in a pre-processing phase, via *query rewriting*, i.e., modifying the user query into an *authorized* one. Indeed, this is one of the most common approaches on controlling access and has several advantages, such as avoiding information disclosure due to visible data manipulations and exploiting existing query optimization techniques. Moreover, it is particularly suited for settings where storage and access control tasks are decoupled, since answers can be safely returned directly to the user who issued the query, without any additional filtering of sensitive data.

We revisit in this paper access control in a distributed setting similar to the one considered in [2], where each source may delegate to other peers some or all of the tasks of defining/enforcing access control for the data it owns. In other words, before being evaluated at the data source, a user query may be modified and restricted by some intermediaries, denoted *access control authorities* (ACAs, in short). The applied restrictions correspond to policies referring to the particular user and data source. A query reaching the source is valid only if certified by all the required ACAs. Once the restricted query is evaluated at the source, the results can be directly returned to the user. This access control setting is illustrated in Figure 2. We refer to this hereafter as *distributed access control*.

Importantly, we consider that access policies may rely on *sensitive information*. The goal of this paper is to contribute

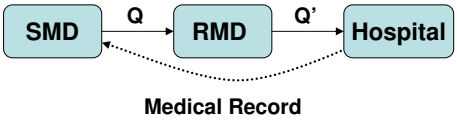


Figure 1: A simple example scenario.

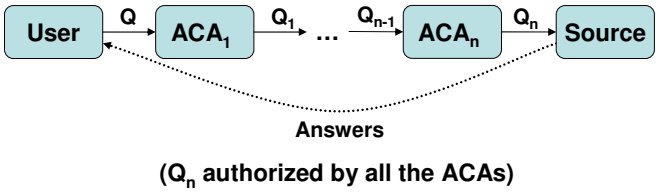


Figure 2: Decoupling access control and resources.

to the understanding of the requirements and challenges that are raised in this context, and to suggest a possible solution for the privacy-preserving implementation of distributed access control. Our solution uses cryptographic techniques, and in particular non-conventional (homomorphic) digital signatures.

The foremost challenge and our main focus is on preventing information leaks that may occur during query rewriting while, at the same time, providing provenance and validity guarantees for queries "traveling" between a user and the targeted data source. As witnessed in Example 1, simply disclosing to either the user, the source or other ACAs that query transformations were performed at some step may breach privacy. We stress that this does not mean that the actual effect of transformations should also be hidden, but only the fact that such operations occurred *after* the user initiated the exchange.

To the best of our knowledge, this is the first paper to address the issue of information leaks that may occur in query rewriting-based access control in non-centralized environments. While the work presented in this paper is a step towards enforcing access control when both data and policies are distributed, more is needed in order to achieve this task without breaching privacy. Other aspects, such as the effect of collusion or the inference of private information from query results (see, for example, [20]), are also important. A complete study of these aspects is beyond the scope of this paper.

Outline. We start by identifying the core issues that need to be addressed in the context of distributed and privacy-conscious access control, in Section 2. In Section 3, we introduce auxiliary cryptographic tools, namely homomorphic signature schemes for modifiable collections. We first illustrate their usage and the main ideas behind our approach by considering relational data and select-project-join queries with non-equalities, in Section 4.1. In Section 4.2, we extend and adapt these techniques to XML data accessed in terms of a rich family of tree pattern queries. We consider related work and we conclude in Section 5.

2. PRELIMINARIES

We start by introducing some initial assumptions and auxiliary notions. We assume some familiarity with common

cryptographic notions such as public-key digital signing (for a comprehensive study we refer the reader to [22]).

Basic assumptions. For simplicity, we assume that the ACAs who need to be first consulted in order to access a given source are always publicly known. We assume that queries follow a linear path, in which each ACA is visited once (as illustrated in Figure 2). (Indeed, this is the only scenario which does not pose a threat to privacy.) Also, the order in which ACAs intercept a query is not important for the source; regardless of the path that was followed, all that matters is which authorities are certifying it as valid. So, when a query is initially sent, the user chooses one of the required ACAs and redirects her query to this authority. Besides the origin and target destination, we assume that each query will carry an ID and a timestamp. Parties are assumed to have well-known identities to which public/private keys are associated. While ACAs are trusted to apply the access policies they are assigned and accordingly, have access to the (potentially private) information on which their policies rely, they should not gain access to any other details about users and sources.

Query rewriting-based access control. Many access control frameworks adopt a query-level enforcement of policies [32, 31, 26, 19, 17], in which all the validity checking is done in a pre-processing phase, before accessing data. In this way, user queries are modified into forms which are authorized (if at all possible), the modifications being aimed at restricting the query to reveal only accessible data.

We assume a role-based approach on access control. Given the user identity and the data item which is to be accessed, each ACA decides to grant or deny access; once access is denied, it cannot be granted back. Denied access translates (if possible) into query restrictions.

Abstracting away from how policies are specified, we will focus directly on the language-specific query restrictions that can be applied to enforce them. According to intuition, what is returned (explicitly or implicitly) by a restricted version of a query should be a subset of what is returned by the initial one. Informally, ACAs can restrict queries by either introducing additional conditions that must be met by the queried data, or by limiting the scope of the query output. We say that a query is *authorized* by an ACA w.r.t. a certain user and source if it obeys the policies handled by that ACA. Importantly, we assume a "monotone" semantics for query-based access, in the sense that if a query is considered authorized by some ACA, then a more restricted version of it should be authorized as well. More details on query restrictions will be given in Sections 4.1 and 4.2, when discussing relational and XML queries respectively.

Distributed Access Control requirements. We next discuss the main requirements that need to be addressed in the context of distributed privacy-preserving access control:

- *Query provenance.* An important aspect of distributed access control in general is the need for *provenance* guarantees. Being able to determine where an incoming query originates, even if it may have been modified along the way, is even more crucial in our case, since access decisions are made based on the identity of the querier. So, a first requirement is that the origin of a query that arrives at an ACA or at the target source must be authentic, in the sense that it should not be possible to maliciously attribute queries to users. Obviously, this is needed when choosing and enforcing access policies, but may also be useful in tasks such as auditing and monitoring access as well.

- *Query validity.* Besides provenance, a source must be able to verify that an incoming query is authorized by the all the required ACAs; only in this case, the query is considered *valid*. Otherwise, the query is rejected. Thus a second requirement is that false validity assertions for queries should be prevented.

- *No history.* As illustrated in Example 1, since access policies may depend on sensitive information, simply disclosing how they were applied in the ACA process, and as a consequence, *how* a given query was modified, may cause a privacy breach. In other words, while intermediaries must certify that a given query is authorized, they have to be able to hide the fact that some query aspects do not originate at the query issuer. This should be hidden to the user, source or even other authorities.

Unsurprisingly, what is particulary problematic and almost contradictory in this model is the need to guarantee provenance and validity for a query, while hiding what transformations occurred at ACAs.

In general, in order to authenticate a piece of information that is exchanged, one usually needs to digitally sign it. Indeed, this can be done by the user who issues the query, in order to prove its provenance. Similarly, if some intermediary (ACA) intercepts a query and wants to certify that the resulting (restricted) version is valid (as far as she is concerned), she can do this by digitally signing it. However, while digital signatures are well established techniques for enforcing data integrity and authenticity, the common assumption is that they disallow any kind of modification on the signed message. Hence, once a query is signed by the user or some ACA, this would prevent downstream ACAs from further restricting it. This means that the query will remain valid (w.r.t. the ACA who signed it) only as long as it is not modified. In fact, what we want is for ACAs to be able to modify an incoming query *provided they only perform restrictions*. In particular, ACAs should not be able to undo or relax previous transformations.

Supporting only partially these requirements is fairly easy. For instance, if we accept to disclose the query changes to the data source, the simplest way would be to take the following high-level steps[1] (see Figure 3(a)):

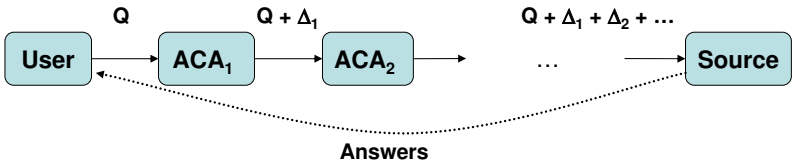
- the user signs her query (using some classic signing scheme) and sends it forward to the intermediaries that have to validate it.

- the query passes by each of those intermediaries in turn, and each of them can add some changes (e.g., to a list of deltas) that are to be considered by the destination.

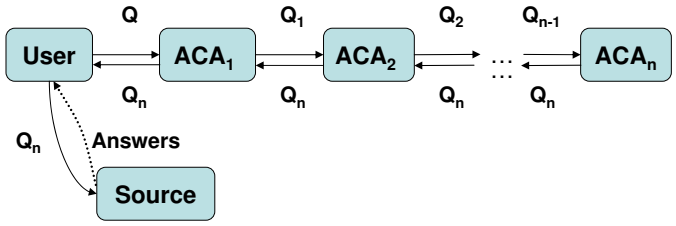
- each of the intermediaries signs her changes (by any classic signing scheme).

- when the query and its list of deltas arrive at the destination, it is immediate to check the provenance of the initial query (still available), of the changes, and the fact that modifications are only restricting the query.

[1]We omit here the straightforward cryptographic details needed for a full-fledged, secure protocol for these tasks.



(a) Disclosing the query modifications to other ACAs and the source by keeping explicit deltas.



(b) Disclosing the query modifications to other ACAs and the user by first obtaining the final query.

Figure 3: Partially supporting the requirements.

Similarly, if we accept to disclose the query changes to the user, the simplest way is by taking the following high-level steps (see Figure 3(b)):

- the user sends her query to the intermediaries that have to validate it.
- the query passes by each of those intermediaries in turn, and each of them performs some changes on the received query.
- the final restricted version follows the same path back, and in the process is signed by each intermediary and finally by the user.
- last, the query and all its certifications are sent to the destination party.

Solution overview. Our solution is based on non-conventional digital signing techniques, which allow some pre-defined type of modifications over signed information. (These are generally known as homomorphic signatures.) To guarantee provenance and to enable validity, the origin will indeed sign her queries, but in a way that allows downstream intermediaries to modify them, provided they only perform restrictions. Each ACA will be able to apply restrictions from a language-specific range, and then update the digital signature of the query (via the mechanisms of homomorphic signatures), reflecting these changes. Since the query signature will remain valid only as long as further restrictions are applied, each ACA can safely put its validity certificate on the query it sends further to the source or other ACAs. Since homomorphic signatures do not reveal anything about the evolution of the signed data, our approach allows ACA authorities to enforce policies by query rewriting while being able to hide all details about the performed modifications. No key sharing is needed and no particular details about potential restrictions to a given query have to be chosen or agreed in advance by the user, source or ACAs. Importantly, our approach is generic, in the sense that it does not depend on how policies are specified, but only on how they are enforced in query rewriting. In this way, any query rewriting-based access control model can rely on our algorithm, using it as a black box.

The following section introduces the cryptographic tools we will use.

3. HOMOMORPHIC SIGNATURES

While the common assumption about digital signatures is that they disallow any kind of modification on signed data, a more flexible approach is often needed and has been advocated lately, one in which some restricted modifications may still occur, without invalidating the data. This is made possible by offering signatures which are *homomorphic* with respect to some operation on the message domain. Starting from the signature(s) of some data instance(s), computed by the data owner, anybody else can derive the signature corresponding to a new data instance, if obtained only via some accepted operation from the previous one(s). More, updated signatures should be *indistinguishable* from the ones computed by the data owner and this updating step should be applicable as many times as needed.

For a given data domain Dom and signature space $\mathcal{S}$, we say that a public-key signature scheme $(Sign_{sk}, Verify_{pk})$ is homomorphic with respect to some binary operation $\odot : Dom^2 \rightarrow Dom$, if there exists an efficient and public *signature update algorithm* $Update_{pk} : Dom^2 \times \mathcal{S}^2 \rightarrow \mathcal{S}$ (or $Update_{pk} : Dom^2 \times \mathcal{S} \rightarrow \mathcal{S}$) such that

$$Update_{pk}(d_1, s_1, d_2, [s_2]) = Sign_{sk}(\odot(d_1, d_2))$$

for any pair of matching public-private keys (pk, sk) and pairs (d_i, s_i) such that $Verify_{pk}(d_i, s_i)$ outputs $\{true\}$.

An important requirement is that signatures obtained via the (secret) signing algorithm and the ones obtained via the (public) updating algorithm should be indistinguishable (as defined below). For such signature schemes, a forgery will refer to a message which is not obtainable via $\odot$ from already signed messages.

Insert-only and delete-only sets. Of particular interest to this paper are two homomorphic set-signing schemes. First, starting from an RSA-like scheme, [25] proposes a public-key set-signing scheme $(Sign_{sk}, Verify_{pk})$ that is homomorphic with respect to both set *difference* and *union*. More specifically, it provides two (public) updating functions $\widetilde{Update}_{pk}$ and $\widehat{Update}_{pk}$, such that:

$$(1)\ \forall U, U' \subseteq U,\ \widetilde{Update}_{pk}(U, Sign_{sk}(U), U') = Sign_{sk}(U')$$

$$(2)\ \forall U_1, U_2,\ \widehat{Update}_{pk}(U_1, Sign_{sk}(U_1), U_2, Sign_{sk}(U_2)) = Sign_{sk}(U_1 \cup U_2)$$

Given a collection $U = \{a_1, ..., a_n\}$ and the signature $s = Sign_{sk}(U)$ (for some secret key sk), someone who wants to remove elements, for instance update U to the subset U', will be able to sign U' by using (1). The signatures obtained by the homomorphic property are completely history independent, because an updated signature is exactly the one that the signing procedure would yield. It is easy to see that we can consider this scheme as one for *delete-only sets* if distinct (fresh) public-private key pairs are used each time a collection is signed.

Second, starting from the above technique for delete-only sets, [3] proposes the dual signature scheme, for *insert-only sets*[2]. More precisely, this scheme provides a public-key set-signing scheme ($Sign_{sk}$, $Verify_{pk}$) and updating function $Update_{pk}$, such that:

$$\forall U, a_{n+1} \notin U,\ Update_{pk}(U, Sign_{sk}(U), a_{n+1}) = Sign_{sk}(U \cup \{a_{n+1}\})$$

Given a set $U = \{a_1, ..., a_n\}$, some new element a_{n+1}, and the signature $s = Sign_{sk}(U)$ (for some secret key sk), someone who wants to insert a_{n+1} can do so without having to know the private key. After insertion(s), she simply updates the set signature. The signatures obtained by the homomorphic property are no longer history independent, because the signing and updating algorithms are probabilistic. However, they are computationally indistinguishable [22]. In particular, the order of insertions is not disclosed.

We stress that the above updating steps on the data and its corresponding signature, for both delete-only and insert-only sets, can be applied as many times as needed.

In practice. Consider someone who wants to publish a collection with one-way updating, allowing everybody else to perform the legal kind of updates. While the signing techniques support this behavior, their success depends on one aspect: the public key used to verify and update signatures. This key should not be modifiable, if we want to guarantee the one-way updating. For that, the owner of the collection has to simply sign this collection key with her (conventional) personal public key. Of course, we assume that each party has such a key, which is also known by everybody else. By signing the collection key we create a "fixed point" from which we can indeed enforce one-way updating. Someone receiving a collection signed in this way, will only be able to conclude that it is either the initial one or a subset (resp. a superset) of the initial one.

[2]The main idea is to use the delete-only technique to enforce a *write-once memory* (i.e., a bit vector). Adding an element to a collection amounts to writing it in the associated bit vector, which in turn amounts to removing some positions from the set of free vector positions.

4. SIGNING MODIFIABLE QUERIES

We are now ready to describe how the two homomorphic signatures described in the previous section can be used to provide authenticity and validity guarantees for queries, without revealing how/if they were modified. For simplicity, low-level details of our cryptography-based approach are omitted.

In the following sections, for relational and XML data in turn, we will detail how queries can be signed such that restrictions can still be performed without invalidating the digital signature. What will be exchanged at each step between the user, ACAs and data source will be a tuple

$$(user, source, Q, S, timestamp, Id, certificates).$$

Q denotes here the actual body of the query while S denotes the modifiable signature of Q (which includes its corresponding Id and *timestamp*), initially computed by the user. *Certificates* denotes a set of tuples

$$(ACA_i, S_i),$$

for S_i being ACA_i's (classic) public-key signature for the tuple

$$(user, source, Id, timestamp).$$

We stress that the query signature is initially computed by the user and then only updated by intermediaries. Each of them certifies Q by signing not the query body, but only its "name". We next detail the techniques for modifiable query signatures.

4.1 Relational Data and Queries

We start by considering relational data and select-project-join queries (also known as conjunctive queries) with non-equalities (denoted $CQ^{\neq}$). We then consider in Section 4.2 XML data and tree pattern queries.

We first illustrate query rewriting within this class by an example. We will assume that queries are first normalized, i.e., joins are made explicit (by variable equalities), with no variable appearing twice in query terms.

EXAMPLE 4.1. *Consider the following* `MedicalRecord` *relational schema:*

patient(ssn, name, address),
visit(vID, ssn, date, wardNo, diagnosis),
treatment(tID,vID, treatment, comments),
diagnosis(name, type)

Consider the following user query Q, which asks for "`Social security number`*,* `diagnosis`*,* `treatment` *and other* `comments`

for patients who were treated on 21/01/2001":

$$(Q)\quad q(S,D,T,C) \;:-\; patient(S,N,A), S=S'\\ visit(V,S',\text{“21/01/2001”},W,D),\\ treatment(I,V',T,C), V=V'$$

And assume that one or several ACAs restrict Q, obtaining in the end the version Q' :

$$(Q')\quad q(S,D,T) \;:-\; patient(S,N,A), S=S'\\ visit(V,S',\text{“21/01/2001”},W,D),\\ treatment(I,V',T,C), V=V'\\ diagnosis(D',T'), D=D',\\ T'\neq \text{“}bloodRelated\text{”}$$

Notice that two modifications occurred. First, the C-variable was projected out of the output tuples. Second, the condition that the `diagnosis` *(D) is not of type* `bloodRelated` *was introduced.*

CQ$^{\neq}$ query rewriting. The $CQ^{\neq}$ query restrictions that we consider are the following:

- add new query terms.
- introduce a join either between existing terms of the query, i.e., an equality between two variables, or between a variable and a constant.
- introduce a non-equality either between two variables which are not already marked as equal, or between a variable and a constant.
- remove a variable from the output.

It is straightforward to check that these transformations will only yield more restricted queries, since the restricted version is contained in the initial one (as a boolean query) and the output variables of the former are among the output variables of the latter.

Next, notice that we can describe a $CQ^{\neq}$ query using several collections (concerning variables, output variables, terms, equalities or non-equalities), as follows:

- C_1, for query terms.
- C_2, for variables.
- C_3, for tuples (query term, variable, position).
- C_4, for pairs of equal variables, or pairs of equal variables and constants.
- C_5, for pairs of non-equal variables, or pairs of non-equal variables and constants.
- C_6, for output variables.

Now, assume that the user digitally signs her initial query using homomorphic signatures for its various collections, as follows:

- C_1, C_2, C_3, C_4, C_5 as insert-only collections.
- C_6 as a delete-only collection.

So, the query signature will consist of seven parts, each of them referring to one of the collections. Obviously, these collections can describe meaningless queries, but this is a situation that can be easily detected and poses no security threat.

If these steps are taken, then any ACA downstream can perform restrictions and then update the query signature accordingly. More specifically, this means that ACAs can only remove output nodes or introduce more conditions (terms, equalities/inequalities). They will be able to do these transformations in a persistent and confidential manner, while modifications that do not further restrict the query are not possible.

EXAMPLE 4.2. *Revisiting Example 4.1, the restrictions can be seen as: (a) removing the C-variable from collection C_6, (b) adding the* `diagnosis` *term and its variables to C_1, C_2 and C3 respectively, (c) adding the equality $D = D'$ to C_4 and (d) adding the non-equality $T' \neq$ "bloodRelated" to C_5.*

We note that for the $CQ^{\neq}$ query language, delete-only sets play a lesser role in our signing approach, and in fact we could rely completely on insert-only ones (although with higher computation cost). However, delete-only sets should become necessary in more expressive SQL-style queries, for example in the presence of disjunction. We omit further details.

4.2 XML Data and Queries

We now employ similar ideas for XML data, the de facto standard for data exchange. In short, an XML document contains nested labeled elements and text, and can be viewed as a rooted, unranked tree with labeled *element* nodes and *text (or data value)* nodes. Figure 4 illustrates an XML document and its corresponding tree representation. For simplicity, we ignore other aspects such as attributes, idrefs or ordering among siblings (more details about XML can be found in [33]). XML data is often associated with schema information (such as DTDs or XSchema) which describes the possible structure of documents. For exemplification, we adopt here the Document Type Definition (in short, DTD) of [19], illustrated in Figure 5, which specifies the structure of medical records. This DTD says that a hospital contains records which are either immediate children of a `medicaRecords` node or are found below intermediary `clinicalTrial` nodes (denoting records from medical trials). A patient record contains various information such as `treatment` and `diagnosis`. Some of the fields of the patient record are considered optional (denoted by the ? annotation). In particular, the document fragment of Figure 4 conforms to this specification.

XML queries. The XML query language that we consider here is essentially an extension of the most commonly used XPath fragment (namely XPath{/,[],//,∗}) with (a) multiple output nodes and (b) negated patterns. By / we denote *child axis* navigation, by // we denote *descendant axis* navigation, and ∗ denotes the wildcard label, which matches any node label. We represent such queries by patterns, as illustrated in Figures 6(a) and 6(b). Edges can be *positive* (representing branches that must be matched) or *negative* (the sub-pattern starting at a negative edge must not match). Moreover, two kinds of output nodes can be specified: nodes which are returned without their corresponding subtree in the document (denoted *simple-output*

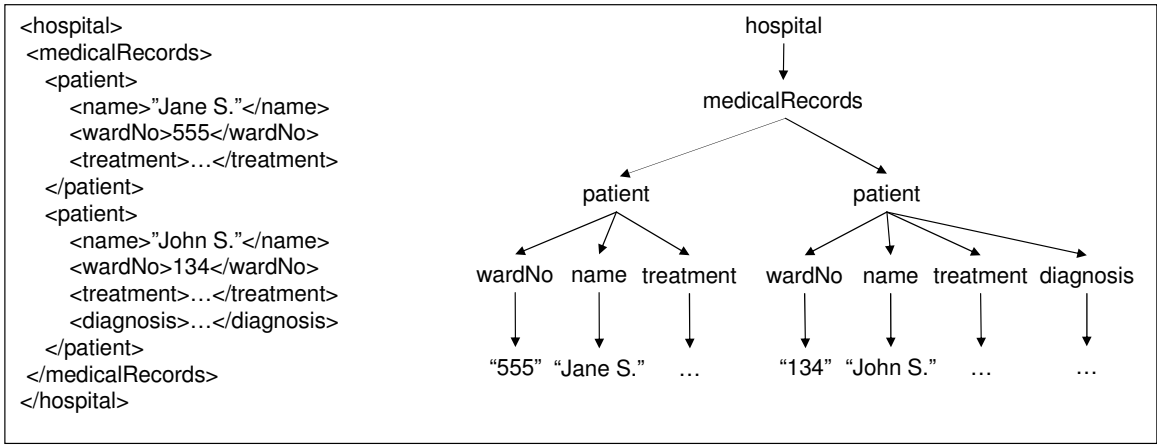


Figure 4: An XML document and its tree representation.

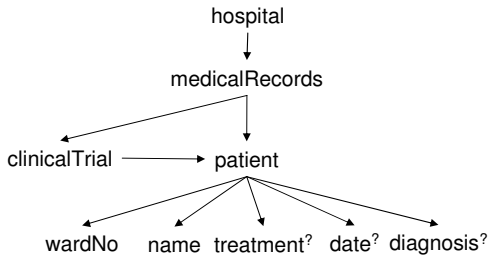


Figure 5: A DTD for medical records.

nodes) and nodes for which we return their entire subtree (denoted *subtree-output* nodes). Nodes which are not returned by the query are called *non-output* nodes.

We use the following graphical representation: simple edges denote child axis, double edges denote descendant axis, simple-output nodes are marked by an s annotation and subtree-output nodes are marked by a t annotation. Negative edges are annotated by $\neg$. The nodes with positive (resp. negative) incoming edges are also called positive (resp. negative) nodes. Obviously, output nodes can only occur in the positive parts of the query. Finally, all the edges / nodes below a negative edge are assumed negative.

Classically, the semantics of a query is the following: on the document tree, we match all the query's positive edges, we check that negative patterns cannot be matched, and the result is the tuple containing the simple-output nodes and the subtrees rooted at subtree-output nodes. We omit here formal definitions of the language and its semantics (see, for example, [29] and [5]). An example of a query and its result over the previous document fragment are given in Figure 6.

XML query rewriting. We assume the following range of modifications that restrict a query:

(a) add anything below a subtree-output node (i.e., positive sub-patters, output nodes and negative sub-patterns).

(b) add a positive edge below a positive edge.

(c) add a negative sub-pattern below a positive edge.

(d) remove the s annotation of a simple-output node (i.e., turning a simple-output node into a non-output one).

(e) transform the t annotation of a subtree-output node into an s annotation (i.e., the subtree rooted at that node is no longer returned).

(f) expand a positive descendant edge into a more detailed (hence less general) linear path.

(g) turn a wildcard label of a positive node into a concrete label.

For instance, the query in Figure 6(b) is a restricted version (after applying some of these transformations) of the one in Figure 6(a). Let us now rephrase the scenario of Example 1 in XML terms:

EXAMPLE 4.3. *Let us assume that the initial query (Q) from the user (`SMD`) asks for the medical data of patient `John Smith`, including (by the descendant axis) that coming from `clinical trials` (Figure 6(a)). All the data below `patient` elements is to be returned.*

Next, this query is restricted by the ACA (`RMD`) into Q' (Figure 6(b)), which has the following differences:

- *it does not access the `clinicalTrial` parts of the record (the descendant axis is transformed into a child one)*
- *it limits the returned data to `treatment` information*
- *and it excludes `bloodRelated` diagnosis from the query result*

For instance, one of the policies that guided the ACA peer in applying these transformations could be the following:

"Physicians who perform `clinical trials` cannot access `clinical trials` of patients with `blood related` diagnosis."

We can easily see that if the query modifications would be known to `SMD` or the `Hospital`, private information would be disclosed. From Q' alone, the `Hospital` cannot understand how the initial query was modified, and from the potential

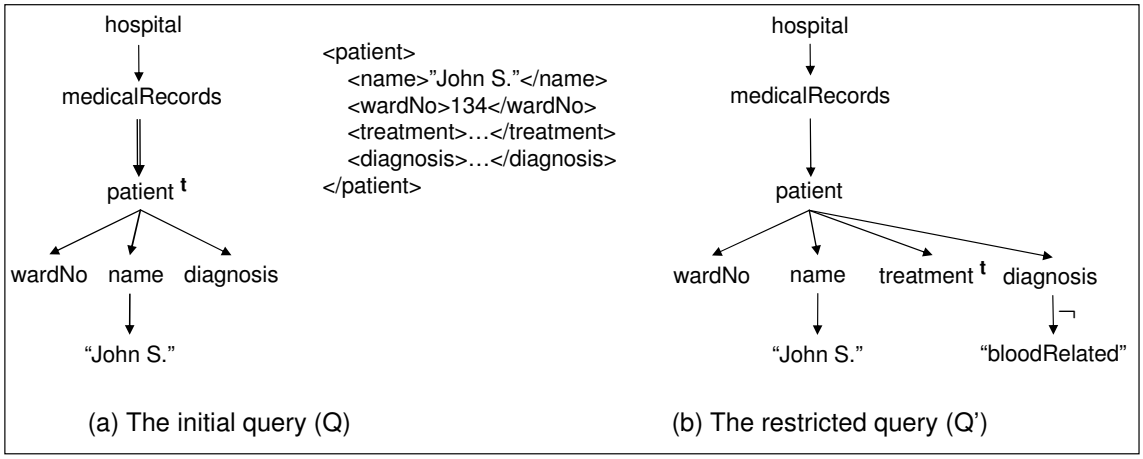


Figure 6: A tree pattern query, its results, and a restricted version of this query.

answers, the SMD peer cannot conclude that there were some `clinical trials` *for that particular patient, or some* `blood related` *diagnosis information.*

We next consider how such restrictions can be handled by modifiable signatures. Similar to the case of relational queries, we need to encode a query into various sets over which we enforce either delete-only or insert-only limitations. While this kind of encoding was straightforward for conjunctive queries, more work is need in the case of tree pattern queries. To this end, we first show how a model of trees with modification restrictions, namely *annotated trees*, can be handled. Then, we show how queries can be translated into this model.

Annotated trees. We extend the delete-only/insert-only notions to trees with nodes annotated by + and/or −, saying that insertions or deletions of children nodes may be prohibited (hereafter called *annotated trees*). More precisely, we consider that node labels are combined with an annotation from +, −, or both (i.e., immutable, noted $\top$). For each node, its annotation will indicate the disallowed transformations on its children set. Intuitively, − (delete-only) for a node indicates that one cannot insert new subtrees as children; + (insert-only) that the deletion of existing children subtrees is not allowed. The absence of annotations indicates the absence of constraints. (An obvious "soundness" rule is that below a unconstrained node everything else is unconstrained.) Besides the tree modifications given by annotations, we also allow for type changes that are further restricting modifications, i.e., adding a new sign.

Next, starting from insert-only and delete-only signatures for sets, we design a signing approach for annotated trees such that one can perform allowed modifications on a signed annotated tree without affecting the authenticity of the tree (i.e., the signature can be updated accordingly) and without revealing the actual modifications. The details of this signing approach are technical but not difficult and are deferred to the Appendix. A similar approach can be employed, for instance, to +/- annotated DAGs.

Encoding queries by annotated trees. The above model takes us one step closer to tree patterns, since queries and the possible restrictions specified for this language can be exactly captured by such annotated trees. We next illustrate the basic steps needed to encode tree pattern queries by annotated trees.

Let us ignore for now descendant edges and wildcard. Figure 7 illustrates how the core blocks of queries can be modeled as a combinations of annotated nodes.

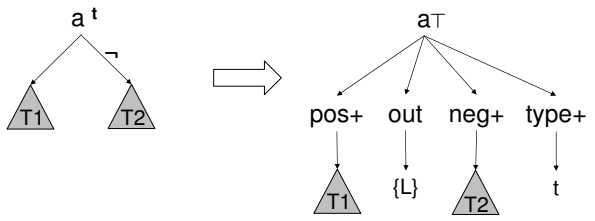
Output properties. Since the output type of a positive node can be modified from subtree-output to simple-output and from there to non-output, we associate to each node an insert-only list in which the reserved labels t, s and n can be added. The labels denote subtree-output, simple-output and non-output nodes respectively. The "weakest" label present in the list will denote the output type.

Subtree-output nodes. In Figure 7(a), we illustrate how a subtree-output node can be represented by an immutable node with four intermediary children. Let L denote the list of output nodes which are children of n, T_1 denote the positive patterns below n and T_2 denote the negative patterns below n.

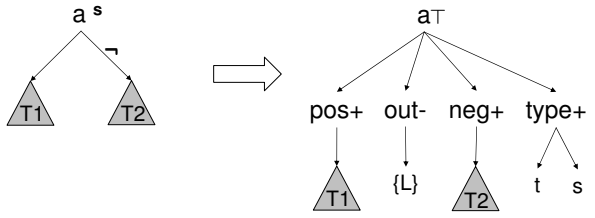
The first three intermediary nodes in the corresponding annotated tree capture these three components (indicated by the *out*, *pos*, and *neg* indices respectively). The fourth annotated node, *type*, denotes the output kind. The node for positive patterns and the one for negative patterns are annotated insert-only (more positive or negative edges may be added), while the one for output nodes is not annotated (for now, output nodes of either type can be both inserted and removed from this list of leaf nodes). Note that output nodes appear both in the insert-only node *pos*, together with their sub-patterns, and in the unconstrained node *out*, as simple leaves.

Simple-output nodes. The case of simple-output nodes is similar (Figure 7(b)). Transforming a subtree-output node into a simple-output one (i.e., moving from Figure 7(a) to Figure 7(b)) amounts to modifying the type of the one unconstrained node (*out*) into delete-only. After this, it is no longer possible to add new output nodes (only non-output ones can still be added), yet removing existing ones can still be done.

Negative nodes. Negative parts are easier then the positive ones, since no modifications are allowed on them. Hence, negative nodes will all be immutable. No intermediary annotated nodes are needed.



(a) Subtree-output node



(b) Simple-output node

Figure 7: Translating queries into annotated trees

Wildcard. This can be easily supported by an overall insert-only list which keeps node/label associations. A node can appear at most once in this list, and one which is not in the list is assumed labeled by wildcard.

Descendant axis. Regarding descendant axis, we need to capture the fact that a //-edge can be transformed into any (more refined) linear path. We can support such transformations as follows:

- for each pair of nodes n_a, n_d such that n_a is an ancestor of n_d, n_d will be among the children of n_a in the annotated tree (i.e., in the *pos* and eventually *out* branches). This means that the annotated tree will degenerate in a DAG. By default, all the edges will now be considered //-edges.
- when we want to express a /-edge or transform a //-edge into a /-edge, we can for instance insert the (parent, child) pair in an insert-only list for the entire tree.
- when a //-edge is expanded, for instance into a longer path, new descendants/ancestors will appear for some of the nodes. This will have to be reflected in their corresponding annotated nodes.

Finally, we stress that we are concerned only with valid query instances; so we ignore transformations that would result in meaningless queries which can be easily detected by analyzing the encoding. We conclude this section by noting that richer query features, such as joins, disjunction or optional edges, may be controlled similarly.

5. CONCLUSION AND RELATED WORK

We considered in this paper the privacy issues which are raised in distributed access control. When access policies are distributed and may depend on private details concerning the various actors involved, previous approaches fail to protect privacy. In particular, information leaks may occur during access control enforcement. Starting from homomorphic set-signing techniques, we suggested a possible implementation for both relational and semi-structured data. We focused primarily on techniques for digitally signing modifiable user queries. Future work includes the implementation and experimental evaluation of these techniques. Cryptographic signatures for sets may also prove useful in a post-query processing phase, for instance when filtering private data from answers. This is the subject of future research.

Privacy in access control has been mainly studied from a different angle, in which the user may not want to disclose her identity and her queries to the source. This is the case in private information retrieval [28, 12] and in the database-as-service paradigm [23, 16]. A good introduction to access control models for XML and tree-like data can be found in [21]. So far, most approaches on XML access control, such as [6, 15], consider a setting where information and access management are highly centralized. While they offer great flexibility in terms of the definition and the enforcement of access rules, they do not provide means to handle non-centralized access policies and multi-party enforcement. Many works (e.g., [19, 11]) considered query rewriting as a solution for enforcing access control over XML data. Also, the secure exchange of modifiable information has been considered before. In [9], the authors introduce a framework for signing XML documents that have also predefined extraction operations. Anyone can filter out information, being able to derive a signature for selected parts of the document (by blinding the rest). Frameworks for secure data exchange in the presence of updates are presented in [7, 35], with the flow of data (partially) defined in advance. For instance, the limited scenario of Figure 3(a) could be supported by them. None of the works above deals with hiding the performed modifications. Data structures that don't yield anything about the history of operations performed on them have been considered in [30].

Acknowledgments. The author would like to thank Tova Milo and Serge Abiteboul for helpful comments.

6. REFERENCES

[1] J. Abendroth and C. D. Jensen. Partial outsourcing: a new paradigm for access control. In *SACMAT*, 2003.

[2] S. Abiteboul, B. Alexe, O. Benjelloun, B. Cautis, I. Fundulaki, T. Milo, and A. Sahuguet. An electronic patient record "on steroids": Distributed, P2P, secure and privacy-conscious. In *VLDB (demo)*, 2004.

[3] S. Abiteboul, B. Cautis, A. Fiat, and T. Milo. Digital signatures for modifiable collections. In *ARES*, 2006.

[4] M. Benedikt and I. Fundulaki. XML subtree queries: Specification and composition. In *DBPL*, 2005.

[5] M. Benedikt and C. Koch. XPath leashed. Unpublished survey, 2006.

[6] E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. *ACM Trans. on Information and System Security*, 5(3):290–331, 2002.

[7] E. Bertino, G. Mella, G. Correndo, and E. Ferrari. An infrastructure for managing secure update operations on XML data. In *SACMAT*, 2003.

[8] E. Bertino, G. Mella, G. Correndo, and E. Ferrari. An infrastructure for managing secure update operations on XML data. In *SACMAT*, 2003.

[9] L. Bull, P. Stanski, and D. M. Squire. Content extraction signatures using XML digital signatures and custom transforms on-demand. In *WWW*, 2003.
[10] B. Carminati. Selective and authentic third-party distribution of XML documents. *IEEE Transactions on Knowledge and Data Engineering*, 16(10), 2004.
[11] S. Cho, S. Amer-Yahia, L. V. S. Lakshmanan, and D. Srivastava. Optimizing the Secure Evaluation of Twig Queries. In *VLDB*, 2002.
[12] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6), 1998.
[13] J. Clark and S. D. (eds.). XML Path Language (XPath) Version 1.0. W3C Recommendation, November 1999. http://www.w3c.org/TR/xpath.
[14] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML Documents. In *EDBT*, 2001.
[15] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. A Fine-Grained Access Control System for XML Documents. *ACM Trans. on Information and System Security*, 5(2):169–202, May 2002.
[16] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *CCS*, 2003.
[17] S. D. C. di Vimercati, S. Marrara, and P. Samarati. An access control model for querying XML data. In *SWS*, 2005.
[18] Health Level Seven. http://www.hl7.org/.
[19] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.
[20] C. Farkas and S. Jajodia. The inference problem: a survey. *SIGKDD Explor. Newsl.*, 4(2), 2002.
[21] I. Fundulaki and M. Marx. Specifying access control policies for XML documents with XPath. In *SACMAT*, 2004.
[22] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2000.
[23] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, 2002.
[24] A. Harrington and C. Jensen. Cryptographic access control in a distributed file system. In *SACMAT*, 2003.
[25] R. Johnson, D. Molnar, D. Song, and D. Wagner. Homomorphic signature schemes. In *CT-RSA*, 2002.
[26] Y. Kanza, A. O. Mendelzon, R. J. Miller, and Z. Zhang. Authorization-transparent access control for XML under the non-Truman model. In *EDBT*, 2006.
[27] Y. Koglin, G. Mella, E. Bertino, and E. Ferrari. An update protocol for XML documents in distributed and cooperative systems. In *ICDCS*, 2005.
[28] E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *FOCS*, 1997.
[29] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.
[30] M. Naor and V. Teague. Anti-persistence: History independent data structures. Cryptology ePrint Archive, Report 2001/036, 2001.
[31] S. Rizvi, A. O. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD Conference*, 2004.
[32] R. Vercammen, J. Hidders, and J. Paredaens. Query translation for xpath-based security views. In *EDBT Workshops*, 2006.
[33] Extensible Markup Language 1.0 (2nd Edition). `http://www.w3.org/TR/REC-xml`.
[34] XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`.
[35] E. B. Y. Koglin, G. Mella and E. Ferrari. An update protocol for XML documents in distributed and cooperative systems. In *ICDCS*, 2005.

APPENDIX

Signing annotated trees. First, one can view an annotated tree as a collection of edges, i.e., pairs $(parent, child)$. So, the same signing technique may be used for trees with homogeneous node types. For trees that mix delete-only and insert-only nodes, one could separate the delete-only regions and insert-only ones and sign each separately. A difficulty is the region *frontiers*, i.e. those nodes that connect a delete-only region to a insert-only one (parent and child have different types).

To deal with the above situation, a frontier node will "carry" the signature for the collection of edges forming the maximal subtree with the same update type. Hence, we need to make persistent (as long as the node exists, since it may be removable) the signing parameters (e.g. the public key) used for that collection. This means that a frontier node n will be represented as a pair (n, PK_n), and be annotated by a signature (verifiable by PK_n).

Observe that the case of an insert-only region T_1 below a delete-only region T_2 needs no specific handling, since the two regions must have been created by the same peer and are thus signed by the same secret function (one signature for each region).

The opposite situation (a delete-only region T_1 below an insert-only region T_2) needs more attention. The signing functions of the two may be different if the root of T_1 has been inserted by a different party than the one who created the root of T_2. When we insert T_1, we insert the public-key of its signature function in the no-remove region, so that it can't be removed. We thus guarantee it cannot be removed unless T_2 itself disappears.

Now consider immutable nodes. They can be viewed as insert-only nodes with an extra attribute that gives the number of its children, i.e., its cardinality. Since this attribute cannot be deleted, one cannot change the cardinality. We conclude the analysis of the four possible types with unconstrained nodes. Since they can be modified at will, they are simply not taken into account when computing signatures.

Finally, consider type changes, from insert-only or delete-only into immutable. The former case is simple. To change a insert-only node to immutable, it suffices to insert a cardinality attribute. One cannot use this trick to turn a delete-only node to immutable, since insertion is not allowed. In this case, one can prepare this change by including in the no insert collection, besides usual elements, pairs (*node*, *cardinality*) for all possible cardinalities (there is a limited number of them for each delete-only node). We can than remove all but one (corresponding the definitive number of children) to make the collection immutable. However, by the above approach, we do not hide the previous type of the now immutable node.