Access Control Policy Translation and Verification within Heterogeneous Data Federations

Gregory Leighton
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
gleighto@ualberta.ca

Denilson Barbosa
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
denilson@ualberta.ca

ABSTRACT
Data federations provide seamless access to multiple heterogeneous and autonomous data sources pertaining to a large organization. As each source database defines its own access control policies for a set of local identities, enforcing such policies across the federation becomes a challenge. In this paper, we first consider the problem of translating existing access control policies defined over source databases in a manner that allows the original semantics to be observed, while becoming applicable across the entire data federation. We show that such a translation is always possible, and provide an algorithm for automating the translation. We then show that verifying that a translated policy obeys the semantics of the original access control policy defined over a source database is intractable, even under restrictive scenarios. Finally, we describe a practical algorithmic framework for translating relational access control policies into their XML equivalent, expressed in the eXtensible Access Control Markup Language.

Categories and Subject Descriptors
F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Computability theory, Computational logic; H.2.7 [Database Management]: Database Administration—Security, integrity, and protection

General Terms
Algorithms, Management, Security, Theory, Verification

Keywords
Access Control, Relational Databases, XML, XACML

1. INTRODUCTION
The standard way of exchanging data across independent databases (or applications) is to use the W3C’s eXtensible Markup Language (XML) [7] as the common data representation format. Conceptually, this is done by defining a mapping of the source database into an agreed-upon XML representation which is then consumed by the target database. The advantages of XML are even more pronounced when the setting above is generalized into a federation of independent databases, maintained by separate organizations (or departments of a single organization), and in which several pairwise mappings exist. Such federations are ever more common among government and large organizations. As an example, consider a state-wide health care governmental agency integrating several large, independently maintained healthcare systems of individual cities into a single federation. Because each city remains administratively independent, the resulting federation forms a heterogeneous system.

XML has proven essential in such settings, and several frameworks for exposing data as XML have been developed within the research community and by industry. For the case of relational data, the mappings exposing XML representations of the data are often referred to as publishing functions. Since XML is universal and ubiquitous, the users of any of the systems in a federation can access any other database in the federation using a common set of tools. In particular queries and/or update operations can be defined in terms of the exposed XML representation of the data using an XML query language (e.g., XQuery [6]), and then be translated into equivalent statements over the original databases. Because publishing functions can be defined between pairs of independent systems, this framework is very generic, capturing various scenarios in data integration (mediators, P2P, etc.).

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

The manual translation of relational ACPs is tedious and error-prone, especially because (1) real-world relational ACPs often consist of hundreds of rules defined over a similarly large number of database objects (i.e., tables, columns, and tuples); (2) the hierarchical and semi-structured nature of XML creates additional complexity, and as one has to take into account factors such as the propagation of access permissions from parent nodes to child nodes; and (3) each source database may define an individual access control policy pertaining to different sets of users, creating the potential for inter-policy conflicts.

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

1.1 Overview

For concreteness, we focus on the standard SQL access control model on the relational side. Without loss of generality, unless stated otherwise, we consider the case of a single XML publishing function. Also, we assume that the XML publishing function is fixed; that is, the goal is to translate the relational ACP for a given XML publishing function. This reflects the reality that often the publishing functions express contracts between sources that are not easy to change.

Our Contributions. In this paper, we address the following issues:

- Expressibility of translated policies: we show that it is always possible to preserve the semantics of arbitrary relational ACPs over XML representations of the data by providing a polynomial-time algorithm for deriving secure publishing functions given such ACPs.
- Verification of translated policies: we discuss the complexity of the testing whether a given secure publishing function enforces a set of relational ACPs, and show this is intractable even for many restricted scenarios.
- Representing translated policies in an XML access control policy expression language: we provide an algorithm for encoding secure publishing functions in the eXtensible Access Control Markup Language (XACML), thereby easing portability and interoperability with other systems.

Organization. The remainder of the paper is organized as follows. In Sec. 2, we provide the necessary background needed for subsequent discussion in the paper. Sec. 3 examines the expressibility problem. Sec. 4 considers methods for verifying the correctness of translated policies. Sec. 5 provides an algorithm for expressing a translated policy as an XACML policy. Sec. 6 summarizes related work, while Sec. 7 concludes the paper and illuminates directions for future work.

2. BACKGROUND

To illustrate the discussion, we refer to the relation depicted in Fig. 1, containing information about patients of a medical clinic.

2.1 Publishing Relational Databases as XML

Viewing an XML document as an ordered, labeled tree allows one to easily visualize the nesting relationship between elements. For example, Fig. 2 depicts one possible manner of representing the database of Fig. 1 as an XML tree. A unique identifier for each tree node is indicated as a superscript of the node’s label.

A publishing function Π specifies how an XML document is created from the contents of a relational database D conforming to schema S. In our model, we represent publishing functions using the formalism of publishing transducers [12]. This model provides a natural basis for our work, as it has proven to be capable of expressing all major XML publishing languages that have been introduced by both industry [1, 24, 25] and academia [3, 14].

Definition 1. A publishing transducer is given by Π = (Q, Σ, q0, δ) where Q is a finite set of states, Σ is a finite alphabet, q0 ∈ Q is the designated start state associated with the root tag r ∈ Σ, and δ is a finite set of transduction rules.

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

Transduction rules have the form

\[(q, a) \rightarrow (q_1, a_1, \phi_1(x_1; y_1)), \ldots, (q_k, a_k, \phi_k(x_k; y_k))\]

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

Publishing transducers are deterministic: for each \((q, a) \in Q \times \Sigma\), there is a unique transduction rule. In the case of the start state \(q_0\) only a single rule \((q_0, r)\) is defined. Additionally, transduction rules may not contain \(q_0\) or \(r\) in their right-hand sides, and the right-hand side for rules of the form \((q, \text{text})\) must be empty, where \text{text} is a reserved symbol in \(\Sigma\) used to indicate a text segment (PCDATA).

Essentially, a publishing transducer Π operates as a finite state machine which, for an input relational database \(D\), generates an XML document tree in a “top-down” manner, using element tags drawn from \(\Sigma\). Each node \(u\) with label \(a \in \Sigma\) in the tree is associated with a register \(Reg_a(u)\), which is used to store a relation of fixed arity. The transduction rule for the current machine state and node label is applied; the query \(\phi_a(x, y)\) is executed on \(D\) and/or \(Reg_a(u)\), and the result is used to produce the child nodes of the current

<table>
<thead>
<tr>
<th>ssn</th>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456789</td>
<td>Carol</td>
<td>31</td>
</tr>
<tr>
<td>197453163</td>
<td>Doug</td>
<td>45</td>
</tr>
</tbody>
</table>

Figure 1: Patient relation.
node $u$. Here, $\bar{x}_i$ designates a set of “group-by” attributes, i.e., the result of $\phi_i$ is partitioned according to these attributes, with each set in the resulting partition contributing a new child node $w$ of $u$ whose register contains all tuples in that partition set. Conversely, all tuples with identical values for all $\bar{y}_i$ attributes create a single child node $w$, and are grouped together within the register associated with $w$.

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

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

**Example 1.** A publishing transducer that constructs the XML document structure depicted by Fig. 2 from an instance of the medical clinic relational schema of Fig. 1 is given by $\Pi = (Q_1, \Sigma_1, q_0, \phi_1)$, where $Q_1 = \{q_0, q_1\}$, $\Sigma_1 = \{\text{age}, \text{name}, \text{patient}, \text{visits}\}$, and $\phi_1$ consists of the set of transduction rules given in Fig. 3.

### 2.2 Access Control for Relational Databases

Most modern relational database management systems employ a discretionary access control model to specify which users have access to particular database objects (tables and columns). In discretionary models, it is up to the owner (typically, the creator) of a database object to determine appropriate access permissions for specific users. In addition, the grantor of a permission can optionally allow the grantee to bestow the permission (or a more restrictive subset thereof) to other users.

The SQL standard includes GRANT and REVOKE statements which are used to construct a discretionary access control policy over a given relational database. Each such statement serves to grant (or revoke) permissions on a set of database objects to one or more users (or subjects). A GRANT statement may optionally contain the WITH GRANT OPTION, which designates that the subject is granted the ability to pass the granted privilege along to other users. In our model, we use the following concise notation to represent the contents of SQL GRANT statements.

$$(q_0, \text{visits}) \rightarrow (q_1, \text{patient}, \phi_1(s, n, a; \emptyset)),$$

where

$$\phi_1(s, n, a) = \text{Patient}(s, n, a)$$

$$(q_1, \text{patient}) \rightarrow (q_1, \text{@ssn}, \phi_2(s; \emptyset)), (q_1, \text{name}, \phi_3(n; \emptyset)),$$

$$(q_1, \text{age}, \phi_5(a; \emptyset)),$$

where

$$\phi_2(s) = \exists n, a \text{Reg}_{\text{patient}}(s, n, a)$$

$$\phi_3(n) = \exists s, a \text{Reg}_{\text{patient}}(s, n, a)$$

$$\phi_5(a) = \exists s, n \text{Reg}_{\text{patient}}(s, n, a)$$

$$(q_1, \text{@ssn}) \rightarrow (q_1, \text{text}, \phi_6(s; \emptyset)),$$

where

$$\phi_6(s) = \text{Reg}_{\text{ssn}}(s)$$

$$(q_1, \text{name}) \rightarrow (q_1, \text{text}, \phi_7(n; \emptyset)),$$

where

$$\phi_7(n) = \text{Reg}_{\text{name}}(n)$$

$$(q_1, \text{age}) \rightarrow (q_1, \text{text}, \phi_8(a; \emptyset)),$$

where

$$\phi_8(a) = \text{Reg}_{\text{age}}(a)$$

**Figure 2:** XML representation of the relation of Fig. 1.

**Figure 3:** Transduction rules for the example publishing transducer.

**Definition 2.** An access control policy rule is a 5-tuple $(s, q, p, o, g)$, where $s$ is a set of users (subjects) drawn from $U$; $q$ is a relational query; $p$ is a set of permissions drawn from the set $\{\text{read}, \text{insert}, \text{delete}, \text{update}\}$; $o$ is a boolean value taking the value true if the grant option has been granted to $s$, and false otherwise; and $g \in U$ indicates the permission grantor. An access control policy (ACP) consists of a set of access control policy rules.

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

**Example 2.** Consider the following SQL ACP over the schema of Fig. 1, with user set $U = \{\text{UserA, UserB, UserC, DBA}\}$:

R1: GRANT select, update ON Patient TO UserB WITH GRANT OPTION

R2: GRANT select ON Patient TO UserB
Relational Schema
Grant = Y
S
ID1
CarolView is a view over the tuples:
the SQL command:
q
language according to the types of local identities
R3: Grant = Y
S
ID1
Grant = Y

2.3 Federated Identities

Each source database defines its access control policies in terms of a closed set of local identities \( U_i \); typically, such identities are individual users or roles. Since such sets often differ from one source to another, a mapping from local identities to a pool of federated identities that are known across the federation is needed. For example, each SQL GRANT statement selects one or more columns within a table, and may therefore be specified using a conjunctive query (CQ). View definition queries in SQL can be much more complex, as they may include features such as nested queries, negation, and union operations. They essentially correspond to the class of first-order (FO) queries.

3. Translating Relational Policies

In this section, we discuss issues and solutions related to translating a pre-existing access control policy defined over

a relational database \( D \) to an equivalent policy defined over an XML representation of \( D \). Our goal is to have a generic access control policy translation procedure that works with various XML access control enforcement mechanisms. We achieve this in two steps. We start by annotating each node of a given XML publishing function \( \Pi' \) with access bitstrings specifying necessary access control restrictions of the objects that are consumed by those nodes; we call this a secure publishing function and denote it by \( \Pi'' \) throughout the paper. In the second step, specific XML access control mechanisms can be derived from \( \Pi'' \) to correctly enforce the original relational access control policy over the mapped data. While different mechanisms have been discussed in the literature, we focus on describing \( \Pi'' \) in terms of the standard eXtensible Access Control Markup Language (XACML) [22].

Fig. 4 illustrates the intended policy translation process. The inputs consist of a relational schema \( S \), an SQL access control policy \( A \) defined over \( S \), a publishing function \( \Pi \), and an identity mapping function \( I \). Provided with such inputs, we wish to derive a secure publishing function \( \Pi'' \) that augments \( \Pi \) by annotating the output XML tree nodes with additional information, allowing \( A \) to be enforced over the XML representation. These annotations are defined in terms of the federated identities, as captured by the identity mapping function \( I \).

Definition 3. Let \( F = \langle f_1, \ldots, f_m \rangle \) be a list of federated identities and \( P = \{ p_1, \ldots, p_n \} \) be a set of grantable permissions. The access bitstring for a database object \( o \),

\[
B_o = (b_{j_1,p_1} b_{j_1,p_2} \cdots b_{j_m,p_1} b_{j_m,p_2} \cdots b_{j_m,p_n} b_{j_m,p_n})
\]

is a string of \( 2mn \) bits which fully specifies the access control policy for over \( o \): bit \( b_{j_i,p} \) is 1 if ID \( j_i \) holds permission \( p_j \) over \( o \), and 0 otherwise, and bit \( b_{j_i,p} \) is 1 only if ID \( j_i \) holds the grant option for permission \( p_j \) over \( o \).

Example 3. Fig. 5 shows an access bitstring indicating that \( ID_1 \) has select and update permissions, both with the grant option, while \( ID_2 \) and \( ID_3 \) each hold a select permission without grant options.

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

Definition 4. A secure publishing transducer \( II'' = (Q, \Sigma, q_0, \delta') \) is a publishing transducer with \( Q, \Sigma, \) and

1 Usually, the access control policy is defined over a specific XML representation of the data, and not the other way around.
Input: Relational schema $S$, access control policy $A$, publishing transducer $Π = (Q, Σ, q₀, δ)$, identity mapping function $I$
Output: A secure publishing transducer $Π′ = (Q, Σ, q₀, δ')

State the algorithm 3.1: Construction of a secure publishing transducer.

1. $δ′ \leftarrow ∅$.
2. $H \leftarrow \text{parseAndSortACP}(A)$;
3. foreach $r \in δ$ do
   4. $\delta' \leftarrow \text{newRule}(r)$;
5. foreach clause in $H$ do
   6. $B \leftarrow 0^{\text{F}}$;
   7. foreach att ∈ atts(clause) do
      8. $B \leftarrow B \cdot \text{SetBit}(entry)$;
   9. $end$
10. condEntries ← $H$.getConditionalEntries(att);
11. if condEntries ≠ ∅ then
12.   $clause' \leftarrow (q, a, B, φ_i)$;
13. else
14.   $r'$.append(clause');
15. $end$
16. endforeach
17. $δ' \leftarrow δ' \cup r' $;
18. $end$

Algorithm 3.1: Construction of a secure publishing transducer.

$q₀$ defined as before, and $δ'$ is a set of transduction rules of the form

$(q, a, B, φ) \rightarrow (q₁, a₁, B₁, φ₁(x₁; y₁)), \ldots, (q_k, a_k, B_k, φ_k(x_k; y_k))$

where $B$ and each $B_i$ are access bitstrings.

The goal is to ensure that the specified relational ACP $A$ is preserved over the published XML document $Π′(D)$. Through slight abuse of the notation, we indicate by $o \in Π(D)$ that a representation of a given database object $o$ is contained in the XML document $Π(D)$ produced by a publishing function $Π$ on the input relational database $D$. By $\text{adom}(Π(D))$, we refer to the active domain of $Π(D)$, that is, the union of database objects formed by evaluating all $φ$ queries appearing in the transduction rules of $Π$.

Definition 5. Let $xview : F \times P \rightarrow \mathcal{P}(Π(D))$ be a function returning the subset of $Π(D)$ which is accessible to a federated ID $f ∈ F$ within the context of an individual permission $p ∈ P$. A secure publishing transducer $Π′$ preserves an ACP $A$ defined over a database $D$ if, for all $ID$s $f ∈ F$ and for each permission $p ∈ P$, the following conditions are in the published XML document $Π'(D)$:

1. (Sufficiency) $∀ o \in \text{accessible}(f, p) : o \in \text{adom}(Π(D))$;
2. (Necessity) $∃ o \in D : o /\not\in \text{accessible}(f, p) \land o \in xview(f, p)$.

Example 4. Let $I$ be an identity mapping function defined as follows: $I([\text{User}] = ID1, I([\text{UserB}] = ID2, and $I([\text{UserC}] = ID3$. For the XML document listed in Fig. 2 and access control policy of Ex. 2, $\text{xview}(ID3, select) = \{1, 2, 3, 4, 5, 6, 7, 8\}$, comprising the root node, and the leftmost patient subtree containing Carol’s record. Note that for each of these nodes, the associated access bitstring has the ninth bit (corresponding to the select permission of ID3) set, while this bit remains unset in the access bitstrings of all remaining nodes.

The following result shows that the proposed solution is applicable to any relational schema, relational access control policy, identity mapping function, and publishing function provided as inputs.

Theorem 1 (Expressibility of Relational ACPS). Given an arbitrary relational schema $S$, an access control policy $A$ defined over a set of users $U$, an identity mapping function $I$, and a publishing transducer $Π$, a secure publishing transducer $Π′$ exists that preserves $A$ over $S$.

Proof. Alg. 3.1 demonstrates a method for converting a publishing transducer into a secure publishing transducer. We now describe how this algorithm works. $A$ is first parsed and a hashtable is constructed from its contents (line 2). In this hashtable, each key is an attribute in $S$, while each key stores two linked lists containing entries of the form $(I(u), p, g, c)$ where $u$ is a user in $U$, $p$ is the applicable permission, $g$ is true if the grant option has been awarded to $u$ over the keyed attribute (and false otherwise), and $c$ specifies a condition on the access granted to $u$ over the keyed database attribute, or takes the value $∅$ if no condition is present. The first linked list contains all unconditional entries, while the second contains the conditional entries (i.e. those for which $c$ is not $∅$).

Next, the transduction rules of the original publishing transducer are examined one-by-one, and transformed into augmented rules of the output secure publishing transducer (lines 3-27). Recall that each transduction rule may have multiple $(q, a, φ)$ clauses on its right-hand side. Lines 5-25 iterate over each such clause; for every attribute appearing in $φ$, the hashtable $H$ is consulted to extract all relevant entries that assign access permissions over that attribute. Lines 8-10 process the unconditional entries for the current attribute; for each such entry, the appropriate bit is set in the bitstring $B$ (i.e., the values of the mapped ID, permission, and grant option fields within the entry are used to determine this bit position). If there are no conditional entries for the current attribute, the augmented clause is constructed by inserting $B$. Otherwise, the conditional entries are then processed one-by-one (lines 17-23). For each such entry, two clauses are constructed: one to handle the case when the condition is not satisfied, and the other to allow additional access to each ID $I(u)$ when the condition $c$ is satisfied. Once all of the clauses within the original transduction rule have been processed in this way, the augmented rule is added to the transduction rule set of the secure publishing transducer (line 26).

To complete the proof, we need to show that both preservation conditions are guaranteed by Alg. 3.1. For the sufficiency condition, observe that if a database object $o \in \text{adom}(Π(D))$ and furthermore, $o \in \text{accessible}(f, p)$ for some federated ID $f$ and permission $p$, then by definition there
must be a policy rule in \( A \) granting \( p \) on \( o \) to some \( u \) for which \( I(u) = f \). In such a case, Alg. 3.1 will set the appropriate permission bit for \( f \) to 1, ensuring that \( f \) holds \( p \) over \( o \). For the minimality condition, note that if \( o \notin \text{accessible}(f,p) \), then there is no policy rule in \( A \) granting access to any \( u \) for which \( I(u) = f \). In such a case, it is easy to verify that in Alg. 3.1 the permission bit for \( f \) will remain 0 regardless of whether \( o \in \text{adom}(\Pi(D)) \), meaning that \( f \) does not have the designated permission over \( o \), and hence \( o \notin \text{xmlview}(f,p) \).

The time complexity of Alg. 3.1 is \( O(|\delta| \cdot |S| \cdot |U|) \), while the storage requirement for the hashtable generated in the first phase is \( O(|S| \cdot |U|) \).

Example 5. We describe the secure publishing transducer that is constructed by Alg. 3.1, using as inputs the publishing transducer from Ex. 1, together with the ACP from Ex. 2. In the first phase, the ACP is parsed as described above, and the following hashtable is constructed:

\[
\begin{align*}
\text{Patient}.\text{ssn} & \rightarrow (\text{ID1}, \text{select}, f, \emptyset), (\text{ID1}, \text{update}, f, \emptyset), (\text{ID2}, \text{select}, f, \emptyset), (\text{ID3}, \text{select}, f, \text{Patient}.\text{name} = \text{Carol}) \\
\text{Patient}.\text{name} & \rightarrow (\text{ID1}, \text{select}, f, \emptyset), (\text{ID1}, \text{update}, f, \emptyset), (\text{ID2}, \text{select}, f, \emptyset), (\text{ID3}, \text{select}, f, \text{Patient}.\text{name} = \text{Carol}) \\
\text{Patient}.\text{age} & \rightarrow (\text{ID1}, \text{select}, f, \emptyset), (\text{ID1}, \text{update}, f, \emptyset), (\text{ID2}, \text{select}, f, \emptyset), (\text{ID3}, \text{select}, f, \text{Patient}.\text{name} = \text{Carol})
\end{align*}
\]

The resulting secure publishing transducer is \( \Pi' = (Q, \Sigma, q_0, \delta') \), with \( Q = \{q_0, q_1\} \), \( \Sigma = \{\#\text{ssn}, \text{name}, \text{age}\} \), and \( \delta' \) consisting of the transduction rules listed in Fig. 6. An example XML tree output by \( \Pi' \) is depicted in Fig. 2.

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

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

\[
\begin{align*}
(q_0, \text{visits}, 111110001000) & \rightarrow (q_1, \text{patient}, 111110000000, f_1(s, n, a; \emptyset)), (q_1, \text{patient}, 111110001000, f_2(s, n, a; \emptyset)), \\
& \text{where } f_1(s, n, a) = \text{Patient}(s, n, a) \wedge n \neq \text{Carol}, f_2(s, n, a) = \text{Patient}(s, n, a) \wedge n = \text{Carol} \\
(q_1, \text{patient}, 111110000000) & \rightarrow (q_1, \@\text{ssn}, 111111000000, f_3(s; \emptyset)), (q_1, \text{name}, 111110000000, f_4(n; \emptyset)), (q_1, \text{age}, 111110000000, f_5(a; \emptyset)), \\
& \text{where } f_3(s) = 2n, a \text{Reg}\text{patient}(s, n, a), f_4(n) = 3s, a \text{Reg}\text{patient}(s, n, a), f_5(a) = 3n, n \text{Reg}\text{patient}(s, n, a) \\
(q_1, \text{patient}, 111110001000) & \rightarrow (q_1, \@\text{ssn}, 111111000100, f_3(s; \emptyset)), (q_1, \text{name}, 111111000100, f_4(n; \emptyset)), (q_1, \text{age}, 111111000100, f_5(a; \emptyset)), \\
& \text{where } f_3(s) = \text{Reg}\@\text{ssn}(s), f_4(n) = \text{Reg}\text{name}(n), f_5(a) = \text{Reg}\text{name}(a) \\
(q_1, \text{patient}, 111111000100) & \rightarrow (q_1, \text{text}, 111111000000, f_6(s; \emptyset)), (q_1, \text{name}, 111111000100, f_7(n; \emptyset)), (q_1, \text{age}, 111111000100, f_8(a; \emptyset)), \\
& \text{where } f_6(s) = \text{Reg}\text{name}(s), f_7(n) = f_6(a) = f_8(a) = \text{Reg}\text{name}(a)
\end{align*}
\]

Figure 6: Transduction rules for the example secure publishing transducer.

4. VERIFYING POLICIES

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

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

We address two variants of the problem: a static analysis ensures that a supplied secure publishing transducer preserves the semantics of a relational ACP \( A \) defined over any instance of relational schema \( S \), while a dynamic analysis takes a specific instance database \( D \) as an additional input and only ensures that the specified secure publish-
Table 1: Complexity of static and dynamic verification of common classes of secure publishing transducers.

<table>
<thead>
<tr>
<th>Fragment</th>
<th>Dynamic Verification</th>
<th>Static Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPT(ℒ, r₁, A)</td>
<td>2EXPTIME</td>
<td>undecidable</td>
</tr>
<tr>
<td>SPT(ℒ, tp, A)</td>
<td>EXPTIME</td>
<td>undecidable</td>
</tr>
<tr>
<td>SPTₙᵣ(F₀, tp, A)</td>
<td>PTIME</td>
<td>undecidable</td>
</tr>
<tr>
<td>SPTₙᵣ(CQ, tp, A)</td>
<td>PTIME</td>
<td>Π²ᵖ-complete</td>
</tr>
</tbody>
</table>

In addition, SPTₙᵣ denotes the more restrictive class of secure publishing transducers that lack recursive transduction rules. Table 1 summarizes the complexity of deciding whether a variant of a class of secure publishing transducers (SPT) specified as SPT(ℒ, S, A), where

- ℒ is the language for the φ queries appearing in transduction rules: CQ (conjunctive queries) or FO (first-order queries);
- S takes the value tp or r₁, indicating whether each node register in the produced XML tree stores a single tuple or a relation; and
- A specifies the complexity of queries appearing in the relational ACP rules: CQ or FO.

Checking the first condition above requires one to decide equivalence between two publishing transducers. As shown in [13], this is undecidable for all classes of transducers except for SPTₙᵣ(CQ, tp, A), in which case deciding equivalence is Π²ᵖ-complete. For the second condition, we consider only this restricted class. Now, the problem consists of checking whether the bitstrings produced by equivalent nodes in Π’ are identical. Again, this approach works for a fixed list of federated users.

5. EXPRESSING POLICIES IN XACML

The eXtensible Access Control Markup Language (XACML) [22] is an OASIS-endorsed standard consisting of both an XML-based declarative policy specification language as well as a processing model that specifies how such policies are interpreted and enforced. Among other benefits, XACML is well-suited for applications in which multiple access control policies must be integrated, and conflicts between such policies must be resolved. In particular, through the specification of rule combining algorithms, an administrator may indicate conflict resolution protocols stating that rules allowing access take precedence over those denying access to the same resource, or vice-versa.

In this section, we provide an algorithm for generating an XACML policy from a secure publishing function Π', conforming to the syntax of the hierarchical resource profile of XACML 2.0 [2]. Note that since the conversion is done entirely at the schema level, this procedure only needs to be carried out once for each Π'. Thereafter, the generated XACML policy will be equally applicable to all generated XML trees Π'(D) formed by applying Π' to any relational database instance D.

**Definition 6.** An XACML policy is a 3-tuple Χ = ({t_d, rca, rs}, where t_d is the target XML document over which the policy applies, rca is a rule combining algorithm indicating how policy rule conflicts are resolved, and rs is a set of policy rules. Each rule in rs is a tuple (su, re, ac, ef, co), where su, re, and ac denote, respectively, the set of subjects, resources, and actions for which the rule applies; ef is the effect of the rule (permit or deny); and co is a set of boolean conditions that serve to restrict the applicability of the rule to cases in which the conditions all evaluate to true. Resources are node sets identified by XPath expressions.

2Ignoring the bitstrings in Π'(D).
The procedure for translating a secure publishing function into an equivalent XACML policy is given by Alg. 5.1. A rule reachability graph (RRG) is first formed from the set of transduction rules $\delta'$ for $\Pi'$ (line 2), where the root node corresponds to the transduction rule defined for the start state and root node label. Furthermore, a directed edge from the node corresponding to transduction rule $\delta_1$ into that of $\delta_2$ indicates that $\delta_1$ contains a reference to $\delta_2$ within a clause $c_i$ in its RHS. Each edge is labeled with the query $\phi_1$ contained in the clause $c_i$ that formed the edge. If $\Pi'$ is recursive, then its rule reachability graph contains at least one cycle. Fig. 7 shows the rule reachability graph for the secure publishing transducer of Ex. 5.

The constructed RRG is then traversed in preorder (lines 3-39). For each encountered node $n$, variables currentPath and bString are updated to refer to, respectively, the XPath path value indicating the location of $n$ relative to the graph’s root node (line 4) and the access bitstring for $n$ (line 5). Additionally (line 6), the variable parentBString is used to store the access bitstring for the parent node of $n$ (in the case of the root node, the zero bitstring is stored).

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

In the next phase, the access bitstrings of $n$ and its parent are compared (lines 10-22). For each bit position in which the two differ, the associated permission perm for that position is determined (line 12), and two groups are formed (lines 13-18): $\text{perm}_{\text{perm}}$ contains the set of federated IDs for whom the corresponding bit positions for perm are set in bString and unset in parentBString, while $\text{deny}_{\text{perm}}$ holds those IDs for whom the corresponding bit positions are unset in bString yet set in parentBString. Once the bitstrings have been entirely processed, sets $\text{perm}_{\text{Group}}$ and $\text{denials}_{\text{Group}}$ serve to store these sets for all permissions in $P$ (lines 19-20). Seeking to minimize the number of constructed policy rules, individual sets in $\text{perm}_{\text{Group}}$ and $\text{denials}_{\text{Group}}$ that contain the same sets of federated IDs are then merged; each set thus formed is then associated with the union of permissions of each original ID set. The resulting sets are designated as mergedPermitsGroup and mergedDenialsGroup (lines 23-26). In the final step (lines 27-38), a separate XACML policy rule is constructed and added to the policy rule set $rs$ for each member of mergedPermitsGroup and mergedDenialsGroup.

Example 6. We illustrate the discussion using the secure publishing transducer of Ex. 5 as an example. The corresponding RRG is depicted in Fig. 7, while the generated policy rule set is shown in Table 2. From the root node’s access bitstring 111110000000, the groups $\text{perm}_{\text{select}} = \{\text{ID1, ID2, ID3}\}$ and $\text{perm}_{\text{update}} = \{\text{ID1}\}$ are created, corresponding to the set bits in the access bitstring. This in turn results in the creation of policy rules $r_1$ and $r_2$ in Table 2. The next node to be visited is $(q_1, \text{name}, 111111000000)$, setting $\text{currentPath} = \text{“/visits/patient”}$. The condition $n \neq \text{“Carol”}$ is resolved by first recording that the referenced attribute $n$ is the second attribute stored in the $\text{Reg}_{\text{patient}}$. Next, the child edges are traversed in breadth-first order; since the query associated with the edge leading into $(q_1, @\text{ssn}, 111110000000)$ does not store $n$ as part of its answer, this edge is disqualified. The query associated with the edge incident to $(q_1, \text{name}, 111110000000)$ does store $n$ within $\text{Reg}_{\text{name}}$, so the temporary stack $s$ is updated to store name, and the edge leading to $(q_1, \text{text}, 111110000000)$ is next followed; since this rule outputs the value of $n$, $s$ is updated to $\text{name/text()}$, and the search terminates. The $\text{currentPath}$ is updated to $\text{“/visits/patient[\text{name/text()]}\text{neg “Carol”}”}$.

The current bitstring $B_r$ differs with that of the parent (root) node only in the ninth bit position, which results in the creation of the set $\text{deny}_{\text{select}} = \{\text{ID3}\}$ and yields the policy rule $r_3$. Traversing the remaining nodes in the RRG does not produce any additional rules.
6. RELATED WORK

Access control models for relational databases. Early discretionary access control models for relational databases [11, 16] heavily influenced the access control model eventually adopted as part of the SQL standard [20]. Mandatory access control models assign a partially-ordered set of security labels to each database object and user. Access is then typically regulated using a "read-down, write-up" policy: users can only modify those objects at a higher security level, and possess read access to objects whose security level is dominated by their own. While an abundant body of research exists on multi-level relational databases employing mandatory access control models, their acceptance by industry has traditionally been limited; the recent introduction of label-based access controls [17, 23] has led to a resurgence in interest. The role-based access control (RBAC) model [15] allows more flexibility by providing an extra level of indirection: instead of hard-coding permissions to individual users, they are instead granted to roles. Users are then assigned to one or more roles, and inherit the permissions of each such role. Role-based features have been added to the most recent version of SQL, yet to date database vendors have provided varying levels of support for these extensions.

XML access control models. Several XML-specific access control models have been proposed in recent years. In a general sense, they can be divided into interactive, view-based approaches which employ a centralized access control module to control access to XML documents (e.g., [5, 9, 18, 19]), and secure publishing approaches (e.g., [4, 8, 21]), which use cryptographic techniques to enforce a designated access control policy over a single, published document. We note that in all these approaches, the focus is on specifying an access control policy over an existing XML document, and no attention has been paid to the issue of propagating a pre-existing access control policy defined over relational data to its XML representation.

XML access control expression languages. While XML access control models define the semantics of protections specified over an XML document tree, such descriptions, in their original form, tend to be somewhat abstract and lack a uniform syntax. To promote interoperability and integration with other applications, and also to facilitate easy deployment and processing of policies, many models also enable their policies to be exported to an XML access control expression language; such languages allow access control policies over XML documents to be specified in a standardized syntax that is understood by many software tools. Typically, these expression languages are themselves XML vocabularies (e.g., [10, 22]). For example, the secure publishing approach of Miklau and Suciu takes an input tree protection and generates a partially-encrypted XML document, where encrypted subtrees are expressed using the W3C recommendation for XML encryption syntax [10].

7. CONCLUSIONS

In this paper, we studied issues related to integrating existing access control policies defined over source databases within larger data federations. We provided an algorithm that allows the translation of relational access control policies to be carried out automatically, and also examined the difficulty of verifying that an existing translation obeys the
original access control policy. To the best of our knowledge, this is the first characterization of the problems associated with the automatic translation of relational ACPs into equivalent ones over XML mappings of relational databases.

There are several areas for future work. As mentioned in Sec. 2, our current solution assigns access permissions over a data object to all local identities mapped to a specific global identity, as long as at least one such local identity holds the access permission within the ACP defined over the originating data source. This places the onus on security administrators to properly define the identity mapping functions in order to ensure that the need to know principle is not violated. Our future work includes investigating strategies for determining the smallest set of global identities needed to ensure that the original access permissions for each local identity are preserved.

One can also examine other formulations of the verification problem; one variation considers scenarios in which the existing translation is specified not as a secure publishing transducer, but rather in a declarative language such as XACML. Another relaxes the assumptions that the list of federated users is fixed (e.g., to account for users being added to and removed from the federation over time). In such cases, the bitstrings of two different secure publishing transducers are incompatible. Thus, both the static and dynamic verification solutions we provide must be changed: instead of directly comparing bitstrings, we must resort to reasoning about the ACPs encoded in each transducer.

While the present paper focuses on cases where access control enforcement is centralized, other possibilities exist, including secure publishing solutions in which access control is implemented over a single copy of an XML document via encryption. Integrating our framework into such applications requires an algorithm for producing partially-encrypted XML documents by applying a secure publishing transducer to an instance database. A final area of investigation includes development of algorithms for minimizing secure publishing transducers, in terms of query sizes and number of transduction rules.

8. REFERENCES