Comparing rule-based policies

P. A. Bonatti F. Mogavero
Università di Napoli Federico II
Via Cinthia, I-80126 Napoli, Italy
bonatti@na.infn.it
fm@fabiomogavero.com

Abstract

Policy comparison is useful for a variety of applications, including policy validation and policy-aware service selection. While policy comparison is somewhat natural for policy languages based on description logics, it becomes rather difficult for rule-based policies. When policies have recursive rules, the problem is in general undecidable. Still most policies require some form of recursion to model—say—subject and object hierarchies, and certificate chains. In this paper, we show how policies with recursion can be compared by adapting query optimization techniques developed for the relational algebra. We prove soundness and completeness of our method, discuss the compatibility of the restrictive assumptions we need w.r.t. our reference application scenarios, and report the results of a preliminary set of experiments to prove the practical applicability of our approach.

1 Introduction and motivations

Rule-based policies play an important role in security and privacy. Standards such as XACML as well as many trust negotiation frameworks and semantically enriched policy languages including TrustBuilder, Cassandra, PeerTrust, the *RT* family, and Protune [14, 11, 2, 10, 5] are rule-based.

In this context, a significant body of work has been devoted to the reasoning tasks involved in policy enforcement and trust negotiation (mainly deduction and abduction). To the best of our knowledge, the problem of *comparing* rule-based policies has not yet been tackled.

Policy comparison underlies several forms of policy verification and compliance checking, for example:

- Checking whether a policy update has strengthened or weakened the policy (by restricting or enlarging the set of authorizations it entails).
- Checking whether a policy (e.g. published by a server) entails another policy independently conceived (e.g. the privacy policy of a user). Possible applications include privacy-driven service selection and policy compliance checking w.r.t. external regulations such as privacy laws.

Checking policy inclusion in a rule-based framework may be computationally complex. Policies are essentially queries mapping a context (such as an XACML context or a negotiation state) to a table of authorizations (whose schema frequently consists of three attributes: *subject*, *object*, and *action*). Note that policy inclusion is *not* equivalent to evaluating first the two policies and then checking the difference between their outputs, because in general the policies cannot be evaluated at comparison time: Context information may be missing for several reasons (e.g., it may change dynamically, like the set of users for example, it may be sensitive and hence unavailable, etc.) Therefore inclusion should hold for *all* possible contexts, which makes policy containment a query containment problem that in general can be undecidable.

To better understand the actual complexity of policy containment let us first identify the expressiveness of our reference rule-based policy languages. Most trust negotiation frameworks, including Cassandra, PeerTrust, RT, and Protune, essentially encode their policies in Datalog (i.e. logic programs without function symbols). Additional features such as atom annotations (Cassandra, PeerTrust) and semistructured objects with attributes (Protune) are only syntactic sugar that can be translated into pure Datalog in linear time. It is not hard to see that the non-procedural fragment of XACML can be turned into Datalog rules, as well (hint: exploit the direct correspondences between XACML and the policy composition algebra introduced in [4]; then apply the translation from algebraic expressions into logic programs defined in the same paper).

Two crucial aspects in Datalog query containment (and hence policy comparison in the aforementioned frame-

works) are whether recursion is allowed and how restricted it is. If unrestricted recursion is allowed then comparison is undecidable [13]. Some restricted forms of recursion (e.g. regular path expressions) can be allowed, but the corresponding approaches to comparison require exponential or even double exponential time [6, 3]. Moreover, specialized optimizations and heuristics for the automata-based techniques adopted in [6] are currently not available, while the approach in [3] needs overly restrictive assumptions on rule syntax. The only approach in NP [9] does not support union which is extremely common and useful in policies (typically to model maximum privilege).

Therefore, in this work we will rather identify alternative restrictions on policy syntax compatible with typical application scenarios, enabling simpler deduction methods, and reducing worst-case complexity. Such restrictions, of course, mainly concern recursion.

First note that XACML policies cannot be recursive—syntax forbids it—however they have comparison operators (such as `double-greater-than`) which are transitively closed. Then a complete Datalog formalization of the policy should have recursive rules for such transitive closures (we will see later examples of valid inclusions which are not recognized if such transitive relations are not appropriately dealt with). The RBAC profile of XACML models inheritance over role hierarchies, which is transitively closed as well.

In trust negotiation frameworks, recursive definitions have been used for modelling:

- authorization inheritance along subject, object and operation hierarchies;
- certificate chains;

and of course—like XACML policies—transitive arithmetic comparison predicates need to be appropriately taken into account during policy comparison.

Similarly, recent approaches to usable policies such as People Finder [8] make use of sets of policy rules for controlled release of the current location of a user based on conjunctive conditions over user groups, location, and time. Groups are obviously structured in a hierarchy; locations are defined by rectangles that may be contained into each other, thereby forming a hierarchy; time is involved in disequations that define sets of intervals.

In this work we show how to deal simultaneously with unions and the above forms of recursion that are special cases of regular path expressions where transitive closure can be applied only to base relations. In this setting we show how to solve policy comparison problems by means of standard techniques based on inclusion mappings, rather than automata techniques. Our fragment's inherent complexity turns out to be in NP, that is, significantly less complex than inclusion in richer decidable fragments of recursive Datalog.

The paper is organized as follows: In Section 2 we introduce the policy language we deal with by means of examples, and point out the limitations of the traditional approach based on containment mappings. In Section 3 we formalize syntax and semantics of the policy language. In Section 4 we show that the restricted syntax introduced in the previous section is not enough to obtain the results we are aiming at. Then in Section 5 we introduce a few more restrictions compatible with our reference scenarios, adapt the standard query containment method and prove the correctness of our approach. Section 6 illustrates the performance of a prototype implementing our containment checking method, and Section 7 closes the paper by summarizing our results and pointing to interesting topics for future research.

2 Examples

As a first example we adopt an access control policy for a virtual bookshop (the same policy used in the demo of our explanation facility Protune-X, see `http://rewerse.net/i2/` → Software). The policy is slightly simplified for the sake of readability (in particular the object oriented syntax supported by Protune is reformulated in standard terms). We use Prolog's convention and let capitalized identifiers denote variables.

The main access control rules for regulating access to a resource `Res` are

$$\texttt{allow}(\texttt{User},\texttt{read},\texttt{Res}) \leftarrow \texttt{public}(\texttt{Res}). \quad (1)$$

$$\texttt{allow}(\texttt{User},\texttt{read},\texttt{Res}) \leftarrow \texttt{auth}(\texttt{User}), \texttt{subscription}(\texttt{User},\texttt{Subs}), \texttt{covers}(\texttt{Subs},\texttt{Res}). \quad (2)$$

$$\texttt{allow}(\texttt{User},\texttt{read},\texttt{Res}) \leftarrow \texttt{id}(\texttt{ID}), \texttt{credit\_card}(\texttt{CC}), \texttt{owner}(\texttt{ID},\texttt{User}), \texttt{owner}(\texttt{CC},\texttt{User}), \texttt{price}(\texttt{Res},\texttt{P}), \texttt{charged}(\texttt{CC},\texttt{P},\texttt{Res}). \quad (3)$$

Rule (1) lets everybody read public resources; rule (2) gives read access when a user has a subscription that covers the requested resource; rule (3) permits to download any resource by paying with a credit card.

Making access control decisions with these rules is analogous to querying the current context, that determines the value of the predicates in the rules' bodies. In this respect, the query output is a 3-ary relation `allow` containing all the authorizations entailed by the policy.

Therefore, in order to check whether a policy update has increased or decreased the set of authorizations, one should check whether for all contexts, the output of the old policy is contained or contains (respectively) the output of the new policy. Analogously, in order to check whether a policy P complies with some more general regulation R, one should verify whether the output of P is contained in the output of R. In summary, policy comparison is essentially a query containment problem.

In our scenario some predicates, called *extensional* or *state predicates*, are not logically defined and are given by the current evaluation context. Predicates such as `public`, `subscription` and `covers` are extensional predicates. On the contrary, some predicates such as `auth` and `credit_card` are *intensional* or *abbreviation predicates*, defined by further rules (constituting a lightweight, rule-based ontology). For example, several forms of authentication are supported. Here are the rules for credential-based authentication and a more traditional method based on passwords:

$$\begin{aligned} \texttt{auth}(\texttt{User}) \leftarrow\ & \texttt{id}(\texttt{ID}), \\ & \texttt{owner}(\texttt{ID},\texttt{User}), \\ & \texttt{type}(\texttt{ID},\texttt{Type}), \\ & \texttt{isa}^{+}(\texttt{Type},\texttt{id\_type}). \end{aligned} \tag{4}$$

$$\begin{aligned} \texttt{auth}(\texttt{User}) \leftarrow\ & \texttt{declaration}(\texttt{D},\texttt{login}), \\ & \texttt{usr}(\texttt{D},\texttt{User}), \\ & \texttt{pwd}(\texttt{D},\texttt{Password}), \\ & \texttt{correct}(\texttt{User},\texttt{Password}). \end{aligned} \tag{5}$$

Rule (4) says a user can be authenticated by submitting a digital ID whose type is an `id_type`.

Note the transitive predicate $\texttt{isa}^{+}$ in rule (4). This formulation presupposes a hierarchy of credential types. The (simple) predicate `isa` encodes direct child-parent links in the hierarchy, and the transitive closure $\texttt{isa}^{+}$ navigates up the hierarchy. Transitive relations like $\texttt{isa}^{+}$ constitute an elementary form of recursion, as they replace recursive rules like

$$\texttt{isa}^{+}(\texttt{X},\texttt{Z}) \leftarrow \texttt{isa}^{+}(\texttt{X},\texttt{Y}), \texttt{isa}^{+}(\texttt{Y},\texttt{Z})\,.$$

Similar encodings can be used for inheriting authorizations along hierarchies of users, objects, and roles.

Some abbreviation predicates such as `credit_card` depend on a notion of valid credential:

$$\begin{aligned} \texttt{valid}(\texttt{Cred}) \leftarrow\ & \texttt{credential}(\texttt{Cred}), \\ & \texttt{public\_key}(\texttt{Cred},\texttt{K}), \\ & \texttt{issuer}(\texttt{Cred},\texttt{CA}), \\ & \texttt{expires}(\texttt{Cred},\texttt{ExpDate}), \\ & \texttt{challenge}(\texttt{K}), \\ & \texttt{trusted\_CA}(\texttt{TCA}), \\ & \texttt{certifies}^{+}(\texttt{TCA},\texttt{CA}), \\ & \texttt{today}(\texttt{Date}), \\ & \texttt{Date} < \texttt{ExpDate}. \end{aligned} \tag{6}$$

Here $\texttt{certifies}(\texttt{CA1},\texttt{CA2})$ is a state predicate that holds when a certification authority `CA1` certifies the public key of `CA2`. The transitive closure $\texttt{certifies}^{+}$ then corresponds to a certificate chain.

As we already pointed out, checking whether a rule-based policy entails another rule-based policy is a query containment problem. In the simplest case (comparison of single rules) a traditional approach to checking containment of a query $Q_1(\vec{t}) \leftarrow B_1$ in $Q_2(\vec{u}) \leftarrow B_2$ consists in searching for a containment mapping from Q_2 to Q_1, that is, a substitution σ such that $\vec{u}\sigma = \vec{t}$ and $B_2\sigma \subseteq B_1$. Clearly, in the presence of transitive closures this approach needs some extension or adaptation. For example, if $\texttt{isa}^{+}$ is replaced with `isa` in rule (4), then a stricter policy is obtained, however it is easy to see that no containment mapping exists between (4) and its modified version.

Another example is inspired by a hotel reservation service. Suppose a valid credit card is required for room reservation. This policy may be expressed with a rule like

$$\begin{aligned} \texttt{allow}(\texttt{User},\texttt{book},\texttt{Room}) \leftarrow\ & \\ & \texttt{credit\_card}(\texttt{C}), \\ & \texttt{expiration}(\texttt{C},\texttt{ExpDate}), \\ & \texttt{today}(\texttt{Now}), \\ & \texttt{ExpDate} > \texttt{Now}. \end{aligned} \tag{7}$$

Note that predicate $>$ is transitively closed (we should automatically replace all of its instances with $>^{+}$).

This policy may be strengthened by requiring the credit card to be valid at arrival time:

$$\begin{aligned} \texttt{allow}(\texttt{User},\texttt{book},\texttt{Room}) \leftarrow\ & \\ & \texttt{credit\_card}(\texttt{C}), \\ & \texttt{expiration}(\texttt{C},\texttt{ExpDate}), \\ & \texttt{today}(\texttt{Now}), \\ & \texttt{booking}(\texttt{User},\texttt{Room},\texttt{ArrivalDate}), \\ & \texttt{ExpDate} > \texttt{ArrivalDate}), \\ & \texttt{ArrivalDate} > \texttt{Now}). \end{aligned} \tag{8}$$

Intuitively, this policy is contained in (7), however it is easy to verify that no containment mapping exists between (7) and (8).

In a forthcoming section, we show how to check containment of queries with transitively closed predicates by first pre-processing the queries and then looking for a standard containment mapping.

3 Policy abstract syntax and semantics

In this section we formally define the new concepts of CT-query and unions of CT-queries, that formalize the policies we deal with in this paper.

Let p range over predicate symbols, t_1, t_2 range over *terms*, that is, variables and constants, and $\vec{t}$ range over vectors of terms. An *extended atom* E is either a (simple) atom $p(\vec{t})$ or a *transitive atom* $q^+(t_1, t_2)$, where q is a binary relation.

Definition 3.1 A *conjunctive transitive query* (*CT-query* for short) is an expression

$$Q(\vec{t}) \leftarrow E_1, \ldots, E_n$$

where each E_i is an extended atom. ∎

As usual we call $Q(\vec{t})$ and $\{E_1, \ldots, E_n\}$ the head and the body of the query, respectively. They will be denoted by $\mathsf{head}(Q)$ and $\mathsf{body}(Q)$. The *arity of* Q is the arity of its head, i.e., the number of its arguments.

Example 3.2 The problem of finding all the credentials $Cred$ of a given type $Type$ can be encoded as the following CT-query: $CredType(Cred, Type) \leftarrow valid(Cred), isa^+(Cred, Type)$.

An *instance* (i.e. a *context*, in policy terms) is a set of (simple) atoms.

Definition 3.3 The *closure* of a set of (possibly nonground, possibly transitive) atoms S, denoted by S^C, is the union of S and the set of all $p^+(t, u)$ such that (t, u) is in the transitive closure of $\{(v, w) \mid p(v, w) \in S \text{ or } p^+(v, w) \in S\}$. With a slight abuse of notation, for all queries $Q(\vec{t}) \leftarrow \vec{E}$ we shall denote by Q^C the query $Q(\vec{t}) \leftarrow \vec{E}^C$. ∎

Example 3.4
Let Q be the query $Q(X, Y) \leftarrow p(X, a), q(a, b), q^+(b, Y)$. Then Q^C is the query $Q(X, Y) \leftarrow p(X, a), p^+(X, a), q(a, b), q^+(a, b), q^+(a, Y), q^+(b, Y)$.

Definition 3.5 The *answer* to a CT-query Q w.r.t. an instance I, denoted by $Ans(Q, I)$, is the set of all $\vec{t}\sigma$ (with σ a variable substitution) such that $\mathsf{head}(Q) = Q(\vec{t})$ and $\mathsf{body}(Q)\sigma \subseteq I^C$. ∎

Example 3.6 Let Q be $Q(X) \leftarrow p(X), q^+(X, Y)$ and $I = \{p(a), p(b), q(b, c)\}$. Here $\vec{t} = X$ and the only σ such that $\mathsf{body}(Q) \subseteq I^C$ maps X on b and Y on c. Therefore, $Ans(Q, I) = \{b\}$.

Definition 3.7 A *union* of CT-queries is an expression $\bigcup_{i=1}^n Q_i$ where each Q_i is a CT-query and all Q_is have the same arity. The *answer* of such a union w.r.t. an instance I, denoted by $Ans(\bigcup_{i=1}^n Q_i, I)$, is $\bigcup_{i=1}^n Ans(Q_i, I)$. ∎

We shall identify $\bigcup_{i=1}^1 Q_1$ with Q_1.

Definition 3.8 Let $\hat{Q}$ and $\hat{Q}'$ be unions of CT-queries. We say $\hat{Q}$ is *contained in* $\hat{Q}'$, in symbols $\hat{Q} \subseteq \hat{Q}'$, if for all instances I, $Ans(\hat{Q}, I) \subseteq Ans(\hat{Q}', I)$. $\hat{Q}$ and $\hat{Q}'$ are *equivalent* if $\hat{Q} \subseteq \hat{Q}'$ and $\hat{Q}' \subseteq \hat{Q}$. ∎

4 Further complexity problems: reasoning by cases

An important property of "traditional" conjunctive queries, from the point of view of implementations and efficiency, is that in order to compare two query unions it suffices to compare the individual queries in the two unions [12]. We are looking for a fragment of recursive queries enjoying the same property. Unfortunately the language we have been dealing with so far is still too general.

Example 4.1 Consider the queries

$$\begin{aligned} Q_1(X, Y) &\leftarrow p(X, Y), q(X, Y). \\ Q_2(X, Y) &\leftarrow p(X, Y), q(X, Z), q(Z, W). \\ Q_3(X, Y) &\leftarrow p(X, Y), q^+(X, Y). \end{aligned}$$

We have $Q_3 \subseteq Q_1 \cup Q_2$, but $Q_3 \not\subseteq Q_1$ and $Q_3 \not\subseteq Q_2$.

The problem turns out to arise from those shared variables like Z that are only used to create chains of binary relations such as $q(X, Z), q(Z, W)$ in the above example.

Problems arise even in comparing single queries.

Example 4.2 Consider the queries

$$\begin{aligned} isa_1(X, Y) &\leftarrow isa(X, Z),\ isa^+(Z, Y) \\ isa_2(X, Y) &\leftarrow isa^+(X, Z),\ isa(Z, Y). \end{aligned}$$

These queries are obviously equivalent but no inclusion mapping exists between them.

Again, the problem is originated by a shared variable Z involved in a chain of binary relations.

According to our experience, policies do not include such chains *per se*, or just to *count* their length. Usually, in the presence of a pair like $q(X, Z), q(Z, W)$ or

$p^+(X,Z)$, $p(Z,Y)$, variable Z is singled out explicitly to test further properties, that is, Z would typically occur in another predicate. We can exploit this property to reduce union comparison to the comparison of individual CT-queries (see *i-safeness* in the next section). The class of queries that can be dealt with efficiently is then further extended, e.g. by exploiting good properties of variables with a single occurrence and what we call *turning points*, that will be introduced in the next section.

5 Policy comparison

Our approach is based on a pretty standard notion of *inclusion mapping* such as those introduced for the relational algebra [7, 1, 12].

Definition 5.1 An *inclusion mapping* from a query $Q_1(\vec{t_1}) \leftarrow \vec{E_1}$ to a query $Q_2(\vec{t_2}) \leftarrow \vec{E_2}$ is a variable substitution σ over Q_1's variables such that $\vec{t_1}\sigma = \vec{t_2}$ and $\vec{E_1}\sigma \subseteq \vec{E_2}$. ∎

In order to prove the completeness of single CT-query comparison for union comparison we shall restrict our attention to what we call *i-safe* source queries.

Definition 5.2 We say that a union of CT-queries $\hat{Q}$ is *inclusion safe* (*i-safe* for short) iff for all queries Q in $\hat{Q}$ and for all variables x occurring as arguments of a binary predicate p occurring in $\mathsf{body}(Q)$, some of the following conditions hold:

1. x occurs in $\mathsf{head}(Q)$.
2. x occurs also as an argument of a predicate $q \neq p$ in $\mathsf{body}(Q)$,
3. x has only one occurrence in $\mathsf{body}(Q)$,
4. x is a *turning point*, that is, in $\mathsf{body}(Q)$ x occurs only as an argument of p^+ and always in the same position (i.e., always as the i-th argument, for some $i \in \{1,2\}$).

∎

Note that the bookshop policy and the hotel policy are i-safe. All the rules reported in Section 2 are i-safe by points 1 and or 2 in the above definition. Point 3 may be helpful in checking that an attribute exists without restricting its value, such as subscriptions in

$$\texttt{customer}(\texttt{C}) \leftarrow \texttt{has\_subscription}(\texttt{C}, \texttt{SomeSubs})\,.$$

Singleton variables can also be used to select non-root and non-leaf nodes in a hierarchy, as in

$$\texttt{non\_leaf}(\texttt{X}) \leftarrow \texttt{isa}(\texttt{SomeY}, \texttt{X})\,.$$

Concerning turning points, they help in reaching parts of the hierarchies that are not above or below the current node, as in the following rule:

$$\texttt{have\_same\_ancestor}(\texttt{X},\texttt{Y}) \leftarrow \texttt{isa}^+(\texttt{X},\texttt{Z}),\ \texttt{isa}^+(\texttt{Y},\texttt{Z})$$

where Z is the turning point (where the path from X to Y along the hierarchy inverts its direction from "up" to "down").

Instead of complicating the notion of inclusion mapping to deal with transitive predicates, we prefer to pre-process queries by normalizing one query and closing the other query under transitive consequences. We shall prove that this approach is correct. Normalization is needed to handle singleton variables (point 3 of i-safeness).

Lemma 5.3 (Normalization) *Let $p(t,x)$ (resp. $p(x,t)$) be an element of $body(Q)$ such that variable x has no other occurrences in Q. Then Q is equivalent to the query obtained by replacing $p(t,x)$ (resp. $p(x,t)$) with $p^+(t,x)$ (resp. $p^+(x,t)$).*

We shall denote by $nrm(Q)$ the (equivalence-preserving) transformation of Q such that all $p(t,x)$ and $p(x,t)$ satisfying the hypothesis of the above lemma are replaced by $p^+(t,x)$ and $p^+(x,t)$, respectively.

Example 5.4 Given $Q(X) \leftarrow p(X,Y)$, we have $nrm(Q) = Q(X) \leftarrow p^+(X,Y)$, because Y has one occurrence in Q.

Clearly, the closure of a query Q is equivalent to Q, too.

Proposition 5.5 *For all CT-queries Q, Q is equivalent to Q^C.*

The next theorem proves that the combination of inclusion mappings and normalization is appropriate to compare individual CT-queries by means of inclusion mappings.

Theorem 5.6 *Let Q_1 and Q_2 be a pair of CT-queries. If Q_1 is i-safe, then $Q_2 \subseteq Q_1$ iff there exists an inclusion mapping from $nrm(Q_1)$ to Q_2^C.*

Proof: Here we report only the most interesting implication, corresponding to the "only if" part of the equivalence.

By, hypothesis $Q_2 \subseteq Q_1$. By Lemmas 5.5 and 5.3, it follows that $Q_2^C \subseteq nrm(Q_1)$, that is for all instances I it holds $Ans(Q_2^C, I) \subseteq Ans(nrm(Q_1), I)$.

Now, let J be the instance built as follows:

- J contains all simple atoms in $\mathsf{body}(Q_2^C)$;
- for all transitive atoms $q^+(v_1,v_2)$ in $\mathsf{body}(Q_2^C)$, J contains $q(v_1,c)$ and $q(c,v_2)$, where c is a new constant not occurring elsewhere;

- nothing else is in J.

Intuitively, by splitting transitive atoms in two atoms we prevent mappings from transitive atoms of Q_1 to non-transitive atoms of Q_2.

Claim 1 The atoms of J^C that do not contain any new constants are exactly the atoms in $\mathsf{body}(Q_2^C)$.

The proof of this claim follows directly from the construction of J and is left to the reader.

Now let $\vec{t}_i$ be the vector of terms in $\mathsf{head}(Q_i)$ ($i = 1, 2$). Note that if σ is the identity function, then $\mathsf{body}(Q_2^C)\sigma \subseteq J^C$, therefore $\vec{t}_2\sigma = \vec{t}_2 \in Ans(Q_2^C, J) \subseteq Ans(nrm(Q_1), J)$. By the last equality, there must be a substitution ρ_J such that $\mathsf{body}(nrm(Q_1))\rho_J \subseteq J^C$ and $\vec{t}_1\rho_J = \vec{t}_2$.

Claim 2 We can always find a ρ_J satisfying the above properties and mapping no variable of Q_1 on a new constant.

To prove this claim, assume that ρ_J maps some variable on some new constants. We are going to show that ρ_J can be transformed into a substitution ρ'_J whose range does not contain any new constant, and such that $\mathsf{body}(nrm(Q_1))\rho'_J \subseteq J^C$ and $\vec{t}_1\rho'_J = \vec{t}_2$.

Let $X\rho'_J = X\rho_J$ if $X\rho_J$ is not a new constant. Next, let X be a variable of Q_1 such that $X\rho_J = c$, where c is one of the new constants. Clearly, X is not in the head $\vec{t}_1$ of Q_1, since $\vec{t}_1\rho_J = \vec{t}_2$ and the constant c cannot appear in the head of Q_2 by definition. Moreover, X cannot appear in two or more different predicates in Q_1 otherwise the inclusion $\mathsf{body}(nrm(Q_1))\rho_J \subseteq J^C$ would not hold, because by construction c occurs only in one predicate in J. If X has a single occurrence in $\mathsf{body}(Q_1)$ then, by the normalization of Q_1, we have that the variable appears in a transitive atom $q^+(v_1, X)$ (resp. $q^+(X, v_2)$) in $\mathsf{body}(nrm(Q_1))$ so, by setting $X\rho'_J = v_2$ (resp. $X\rho'_J = v_1$), we have $q^+(v_1, X) \in J^C$ (resp. $q^+(X, v_2) \in J^C$), that is, the modified substitution ρ'_J preserves the inclusion of $\mathsf{body}(nrm(Q_1))$ in J^C. Analogously, if X is a turning point for a predicate q^+ in Q_1 at the position $i \in \{1, 2\}$, we can set $X\rho'_J = v_i$ without affecting the inclusion of $\mathsf{body}(nrm(Q_1))$ in J^C.

Summarizing, ρ'_J does not map any term to a new constant and $\mathsf{body}(nrm(Q_1))\rho'_J \in J^C$. Moreover, ρ'_J agrees with ρ_J on all variables that are not mapped on a new constant. These include all variables in the head (see above), therefore $\vec{t}_1\rho'_J = \vec{t}_1\rho_J = \vec{t}_2$. This completes the proof of Claim 2.

Now, since we can choose a ρ_J that maps no variable of Q_1 on a new constant, the inclusion $\mathsf{body}(nrm(Q_1))\rho_J \subseteq J^C$ and Claim 1 imply that $\mathsf{body}(nrm(Q_1))\rho_J \subseteq \mathsf{body}(Q_2^C)$ and hence ρ_J is an inclusion mapping from $nrm(Q_1)$ to Q_2^C. ∎

Example 5.7 This approach to query comparison deals correctly with transitive predicates. Consider again policy rules (7) and (8). The closure of (8) contains `ExpDate` $>$ `Now` (recall that $>$ is always implicitly replaced by $>^+$). This makes it possible to find an inclusion mapping from the normalization of (7) to the closure of (8), and the inclusion between the two policy rules is correctly discovered.

The next theorem shows how to deal with query unions. It shows that unions can be compared by comparing their individual members, as required.

Theorem 5.8 *Let $\hat{Q} = \bigcup_{i=1}^m Q_i$ and $\hat{Q}' = \bigcup_{j=1}^n Q'_i$ be unions of CT-queries. If $\hat{Q}'$ is i-safe, then $\hat{Q}$ is contained in $\hat{Q}'$ iff each Q_i is contained in some Q'_j.*

These theorems provide the main justification for our method: Query unions can be compared by finding inclusion mappings between normalized and closed versions of their members.

Corollary 5.9 *Let $\hat{Q} = \bigcup_{i=1}^m Q_i$ and $\hat{Q}' = \bigcup_{j=1}^n Q'_i$ be unions of CT-queries. If $\hat{Q}'$ is i-safe, then $\hat{Q}$ is contained in $\hat{Q}'$ iff for all Q_i there exists some Q'_j such that an inclusion mapping exists from $nrm(Q'_j)$ to Q_i^C.*

In general, finding an inclusion mapping is NP-complete [7]. Now, by the previous corollary, we have that comparing union of i-safe CT-queries is reducible to the problem of finding an inclusion mapping between pairs of normalized CT-queries and vice versa, where normalization is in PTIME. Then, the following result holds.

Corollary 5.10 *Let $\hat{Q} = Q \bigcup_{i=1}^m Q_i$ and $\hat{Q}' = \bigcup_{j=1}^n Q'_i$ be unions of CT-queries. If $\hat{Q}'$ is i-safe, then the decision problem "Is $\hat{Q}$ contained in $\hat{Q}'$?" is NP-complete.*

6 Experimental evaluation

We conducted two series of experiments to provide preliminary evidence of the applicability of the theoretical approach described in the previous sections.

Safeness checking, normalization, closure, and mapping construction have been implemented in XSB Prolog and executed on a notebook with dual core and 2GB RAM.

The first experiments are based on realistic policies, derived from the bookshop scenario. The base policy has been compared with analogous policies obtained by relaxing and strengthening requirements such as the possible forms of authentication and payment, the set of accepted cards, and so on. Abbreviation predicates (which are not directly supported in the formal comparison framework) are eliminated by unfolding rules (i.e., replacing subgoals with the body of matching rules until only extensional predicates remain).

Even if the bookshop policies are more complex than most of the policies actually enforced in today's systems, their size is small in several respects: the number of CT-queries in each union is about 10, the average number of literals in CT-query bodies is 6, and the maximum number is 14. The entire process of inclusion checking, comprising unfolding, safeness checking, normalization, and body closure, is completed in 0.08 seconds (average time) and never exceeds 0.09 seconds.

More experiments comparing a user's release policy and a server's access control policy (relevant to privacy-aware service selection) confirmed these times.

Given the small size of these policies, we conducted harder, worst case experiments. This second set of experiments is hard (and artificial) because of the following features:

- All CT-queries have the same head (while real policies usually have different targets, which facilitate fast selection of relevant queries for the mapping).
- Each body is a long chain of binary predicates $p(X_1, X_2), \ldots, p(X_{n-1}, X_n)$, with opposite order in the two queries to be compared. The implemented search strategy for inclusion mappings in this case has no clue to focus literal matchmaking and backtracks $n!$ times.
- Inclusion mapping search is repeated m^2 times, where m is the number of CT-queries in each union, as if the right target query for the mapping were always the last one selected by the strategy.

The results (in seconds) are reported in Figure 1. Note that despite extremely unfavorable conditions, real time policy comparison is possible for policies whose size is significantly larger than the bookshop policies. Policy validation (that does not pose any real time requirements) is still possible in a handful of minutes for policies with 250 CT-queries per union and up to 50 literals per body (organized in the artificial chains mentioned before). Realistic policies with similar size are expected to yield significantly shorter processing times, because a predicate frequently occurs at most once in a rule body, and rarely occurs more than twice.

7 Conclusion and future work

Rule-based policy comparison can be carried out by means of a technique (inclusion mapping) originally introduced in the area of database query optimization and extended in this paper to support the forms of recursion commonly needed in security and privacy policies. The initial experiments are encouraging: A simple implementation with no particular optimizations is able to compare complex queries in acceptable time, even under artificially hard conditions.

We proved our techniques to be formally sound and complete. Results and algorithms can be immediately extended to policies written in any rule-based policy language, provided that they satisfy i-safeness and use recursion only to compute the transitive closure of extensional binary relations. The restrictions introduced for completeness appear to be compatible with application needs. Still there is space for further developments.

We plan to give proper support to numeric domains with arithmetic comparison. Currently, the system does not know—say—that 2 is smaller than 17 and $1 + 3$, i.e., the comparison engine is using an *incomplete formalization* of the numeric domain, unaware of numeric constants and arithmetic operators (therefore in the presence of numeric constants or arithmetic operators the current method is incomplete). Arithmetic operators can be supported by integrating the inclusion mapping method with a constraint solver.

We are also planning to extend the framework to support the limited form of negation commonly adopted in security and privacy policies. Other topics for further investigations include subclasses of queries that can be compared in polynomial time, and automata based approaches that have been proved to be complete for richer languages. As we pointed out in the introduction, these techniques are too complex in their generality and need to be restricted to find an acceptable tradeoff between language expressiveness and the complexity of the comparison problem.

Acknowledgements

This work has been partially supported by the network of excellence REWERSE, IST-2004-506779.

References

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

[2] M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, Yorktown Heights, June 2004.

[3] P. A. Bonatti. On the decidability of containment of recursive datalog queries - preliminary report. In A. Deutsch, editor, *PODS*, pages 297–306. ACM, 2004.

[4] P. A. Bonatti, S. D. C. di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35, 2002.

[5] P. A. Bonatti and D. Olmedilla. Driving and monitoring provisional trust negotiation with metapolicies. In *IEEE 6th*

	N rules								
Body len	*10*	*20*	*30*	*40*	*50*	*100*	*150*	*200*	*250*
10	.05	.08	.17	.27	.39	1.44	3.22	5.57	8.64
20	.12	.33	.60	1.01	1.54	6.14	13.70	23.84	37.33
30	.25	.76	1.61	2.88	4.45	16.99	39.00	68.80	108.17
40	.43	1.59	3.47	6.10	9.32	37.46	84.36	150.98	234.45
50	.76	2.88	6.49	11.50	17.63	70.63	161.92	279.65	442.37

Figure 1. Worst case results

International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), Stockholm, Sweden, June 2005.

[6] D. Calvanese, G. D. Giacomo, and M. Y. Vardi. Decidable containment of recursive queries. *Theor. Comput. Sci.*, 336(1):33–56, 2005.

[7] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. Ninth Annual ACM Symp. on Theory of Computing*, pages 77–90, 1976.

[8] Cornwell et al. User-controllable security and privacy for pervasive computing. In *The 8th IEEE Workshop on Mobile Computinf Systems and Applications (HotMobile 2007)*, 2007.

[9] D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, pages 139–148. ACM Press, 1998.

[10] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, and M. Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *1st First European Semantic Web Symposium*, Heraklion, Greece, May 2004.

[11] N. Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 2002.

[12] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *JACM*, 27(4):633–655, 1980.

[13] O. Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, 1993.

[14] T. Yu, M. Winslett, and K. Seamons. Interoperable strategies in automated trust negotiation. In *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 146–155. ACM Press, 2001.