

Efficient Comparison of Enterprise Privacy Policies

Michael Backes, Walid Bagga, Günter Karjoth, Matthias Schunter
IBM Research, Zurich Research Laboratory,
Säumerstrasse 4, 8803 Rüschlikon, Switzerland
{mbc, moh, gka, mts}@zurich.ibm.com

September 12, 2003

Abstract

Enterprise privacy enforcement allows enterprises to internally enforce a privacy policy that the enterprise has decided to comply to, often reflecting different legal regulations, promises made to customers, as well as more restrictive internal practices of the enterprise. The notion of policy refinement is fundamental for privacy policies, as it allows to check whether a company's policy fulfills regulations or adheres to standards set by customer organizations, to realize the "sticky policy paradigm" that addresses transferring data from one realm to another in a privacy-preserving way, and much more. Although well-established in theory, the problem of how to efficiently check whether one policy refines another has been left open in the privacy policy literature. We present a practical algorithm for this task, concentrating on those aspects that make refinement of privacy policies more difficult than, e.g., refinement for access control policies, like a more sophisticated treatment of deny-rules and a suitable way for dealing with obligations and conditions on context information.

1 Introduction

An increasing number of enterprises make privacy promises to customers or, at least in the US and Canada, fall under new privacy regulations. To ensure adherence to these promises and regulations, enterprise privacy technologies are emerging [7]. An important tool for enterprise privacy enforcement is formalized enterprise privacy policies [8, 11, 10, 4]. An enterprise privacy policy often reflects different legal regulations, promises made to customers, as well as more restrictive internal practices of the enterprise. Further, it may allow customer preferences. Compared with the well-known language P3P [12] intended for privacy promises to customers, languages for the internal privacy practices of enterprises and for technical privacy enforcement must offer more possibilities for fine-grained distinction of users, purposes, etc., as well as a clearer semantics.

The notion of policy refinement is fundamental for many situations in privacy policy management. Intuitively, one policy refines another if using the first policy automatically also fulfills the second policy. For instance, policy refinement enables verification that an enterprise policy fulfills regulations or adheres to standards set by consumer organizations or a self-regulatory body, assuming only that these coarser requirements are once and for all also formalized as a privacy policy. Similarly, it enables verification that a detailed policy for a part of the enterprise (defined by responsibility or by technology) refines the overall privacy policy set by the company's CPO. The verification can be done in the enterprise or by external auditors, such as [14]. Sticky policies [11] are another application of policy refinement: With increasingly dynamic e-business, data will be exchanged between enterprises, and enterprise boundaries change due to mergers, acquisitions, or virtual enterprises. After transferring data from the realm of one policy into another (where the transfer must of course be permitted by the first policy), the second realm must enforce the first policy. However, the enforcement mechanisms (both organizational and technical) in the second realm will often not be able to deal with arbitrary policies for each obtained set of data. In this case, one realm must perform a refinement test before the data are transferred, i.e., one has to verify that the policy of the second realm refines the policy of the first, at least for the restriction of the first policy to the data types being transferred. This requires compatible enterprise privacy enforcement mechanisms. For these reasons, IBM has recently proposed an Enterprise Privacy Authorization Language (EPAL) [1] as an XML specification for public comments and possible subsequent input to standardization.

Although refinement of privacy policies is well-established in theory [4], an efficient algorithmic solution for checking if one policy refines another has not been addressed yet. Coming up with such a solution is challenging for three crucial reasons: First, compared to typical access control policies, privacy policies additionally offer a more sophisticated semantics for requests to abstract elements, e.g., an abstract user “department” that is used to group a set of concrete “employees”. Requests to such abstract elements are interpreted in an access control manner, i.e., if the department has at least one employee that is not allowed to perform a specific action, then so is the department as an abstract user. In the representation of the semantics of privacy policies, this formally means that deny-rules have to be inherited upwards the hierarchies. Secondly, obligations as well as conditions on context information have to be taken into account, which are essential features in enterprise privacy policies. Thirdly, a one-to-one adoption of the definition of policy refinement requires to compare each element of the first policy with each element of the second one. Since rules usually overlap for a large number of such elements, an efficient algorithm ought to identify these elements and compared them as a whole.

The goal of this article is therefore to provide an efficient algorithm for checking refinement of privacy policies in an enterprise. We do this concretely for the IBM EPAL proposal. However, for a scientific paper we cannot use the lengthy XML syntax, but have to use a corresponding abstract syntax, which closely resembles the one presented in [4] (which, like EPAL, is based on [11]).

Further related literature. The core contribution of new privacy-policy languages [8, 11, 10], compared with other access-control languages, is the notion of purpose and purpose-bound collection of data, which is essential to privacy legislation. Other necessary features that prevent enterprises from simply using their existing access-control systems are obligations and conditions on context information. Individually, these features were also considered in recent literature on access control, e.g., purpose hierarchies in [6], obligations in [5, 9, 13], and conditions on context information in [15].

2 Syntax, Semantics, and Refinement of EPAL Policies

In this section, we review the abstract syntax and semantics of IBM’s EPAL privacy policy language [2], which closely resembles a recently proposed abstract syntax and semantics for the superset of E-P3P Enterprise Privacy Policies in [4]. The main differences are that EPAL does not use rules with priorities as considered in [4] but the simpler representation as an ordered list of rules, and that EPAL policies are additionally equipped with a global condition that has to be satisfied in order to further process a request, as well as with a default obligation.

2.1 Hierarchies, Obligations, and Conditions

For conveniently specifying rules, the data, users, etc. are categorized in EPAL as in many access-control languages. This also applies to the purposes. To allow structured rules with exceptions, categories are ordered in hierarchies; mathematically they are forests, i.e., multiple trees. For instance a user “company” may group several “departments”, each containing several “employees”. The enterprise can then write rules for the whole “company” with exceptions for some “departments”.

Definition 2.1 (Hierarchy) A hierarchy is pair $(H, >_H)$ of a finite set H and a transitive, non-reflexive relation $>_H \subseteq H \times H$, where every $h \in H$ has at most one immediate predecessor (parent). As usual we write \geq_H for the reflexive closure. We write $h \gtrsim_H h'$ if $h \geq_H h'$ or $h' \geq_H h$ holds.

For two hierarchies $(H, >_H)$ and $(G, >_G)$, we define

$$\begin{aligned} (H, >_H) \subseteq (G, >_G) & \quad :\Leftrightarrow \quad (H \subseteq G) \wedge (>_H \subseteq >_G); \\ (H, >_H) \cup (G, >_G) & \quad := \quad (H \cup G, (>_H \cup >_G)^*); \end{aligned}$$

where $*$ denotes the transitive closure. Note that a hierarchy union is not always a hierarchy again. ◇

Throughout this paper we often speak of hierarchies as forests, i.e., as set of trees.

EPAL policies can impose obligations, i.e., duties for the enterprise. Examples are to send a notification to the data subject after each emergency access to medical data, or to delete data after a given time. Obligations are not structured

in hierarchies, but by an implication relation. As multiple obligations may imply more than each one individually, we define the implication (which must also be realized in the application domain) on these sets. We also define how this relation interacts with vocabulary extensions.

Definition 2.2 (Obligation Model) An obligation model is a pair (O, \rightarrow_O) of a set O and a relation $\rightarrow_O \subseteq \mathfrak{P}(O) \times \mathfrak{P}(O)$, spoken implies, on the powerset of O , where $\bar{o}_1 \rightarrow_O \bar{o}_2$ for all $\bar{o}_2 \subseteq \bar{o}_1$, i.e., fulfilling a set of obligations implies fulfilling all sub-sets.

For $O' \supset \mathfrak{P}(O)$, we extend the implication to $O' \times \mathfrak{P}(O)$ by $((\bar{o}_1 \rightarrow_O \bar{o}_2) :\Leftrightarrow (\bar{o}_1 \cap \mathfrak{P}(O) \rightarrow_O \bar{o}_2))$. \diamond

The decision formalized by a privacy policy can depend on context data. Examples are a person's age or opt-in consent. In EPAL, this is represented by conditions over data in so-called containers [2]. The XML representation of the formulas is taken from [15], which corresponds to a predicate logic without quantifiers. Similar to [4], we formalize the containers as a set of variables with domains, and the conditions as formulas over these variables.

Definition 2.3 (Condition Vocabulary) A condition vocabulary is a pair $Var = (V, Scope)$ of a finite set V and a function assigning every $x \in V$, called a variable, a set $Scope(x)$, called its scope.

Two condition vocabularies $Var_1 = (V_1, Scope_1)$, $Var_2 = (V_2, Scope_2)$ are compatible if $Scope_1(x) = Scope_2(x)$ for all $x \in V_1 \cap V_2$. For that case, we define their union by $Var_1 \cup Var_2 := (V_1 \cup V_2, Scope_1 \cup Scope_2)$. \diamond

In this paper, we will not extend this to a full signature in the sense of logic, i.e., including predicate and function symbols, but we assume a given universe of predicates and functions with fixed domains and semantics. For a condition vocabulary $Var = (V, Scope)$ and for the assume universe of predicates and functions, we let $C(Var)$ denote the set of correctly typed formulas over V . Furthermore, let $\mathfrak{Ass}(Var)$ denote the set of all assignments for the set V into the respective scopes, and for $\chi \in \mathfrak{Ass}(Var)$, let $eval_\chi: C(Var) \rightarrow \{true, false\}$ denote the evaluation function for conditions given this variable assignment. This is defined by the underlying logic and the assumption that all predicate and function symbols come with a fixed semantics.

For an efficient algorithmic solution of policy refinement, we will see that it turns out to be crucial to check whether one condition c_1 satisfies another one c_2 , i.e., whether $eval_\chi(c_1) = true$ implies $eval_\chi(c_2) = true$ for every assignment χ . However, since this problem is NP-complete in the number of variable of the consider condition vocabulary, we cannot expect to solve this for all instances. For practical purposes, one therefore restricts its attention to a *satisfy relation* which is at least *correct*, i.e., if c_1 and c_2 are contained in the relation then $eval_\chi(c_1) = true$ implies $eval_\chi(c_2) = true$.

Definition 2.4 (Satisfy Relation) Let Var be a condition vocabulary. A satisfy relation for Var is a relation $\Rightarrow_{Var} \subseteq C(Var) \times C(Var)$. The relation is correct if for any $c_1, c_2 \in C(Var)$, we have $(c_1, c_2) \in \Rightarrow_{Var}$ only if $(eval_\chi(c_1) = true) \Rightarrow (eval_\chi(c_2) = true)$ for all $\chi \in \mathfrak{Ass}(Var)$. If the converse direction holds, we call the relation complete. In the following, we use infix notation for the relation \Rightarrow_{Var} and we omit the subscript Var if it is clear from the context. \diamond

For practical purposes, a suitable satisfy relation which is correct but not necessarily complete can often be constructed by means of symbolic evaluation [?].

2.2 Syntax of EPAL Policies

An EPAL policy consists of a vocabulary, a list of authorization rules, a global condition, and a default ruling. The vocabulary defines element hierarchies for data, purposes, users, and actions, as well as the obligation model and the condition vocabulary. Data, users and actions are as in most access control policies, and purposes are an important additional hierarchy for the purpose binding of collected data.

Definition 2.5 (Vocabulary) A vocabulary is a tuple $Voc = (UH, DH, PH, AH, Var, OM)$ where UH, DH, PH , and AH are hierarchies called user, data, purpose, and action hierarchy, respectively, Var is a condition vocabulary, and OM an obligation model. \diamond

As a naming convention, we assume that the components of a vocabulary called Voc are always called as in Definition 2.5 with $UH = (U, >_U)$, $DH = (D, >_D)$, $PH = (P, >_P)$, $AH = (A, >_A)$, $Var = (V, Scope)$, and $OM = (O, \rightarrow_O)$, except if explicitly stated otherwise. In a vocabulary called Voc_i all components also get a subscript i , and similarly for superscripts.

The list of authorization rules, short *rule list*, contains rules that allow or deny operations. A rule basically consists of one element from each of the considered hierarchies, a ruling, a condition, and an obligation.

Definition 2.6 (Rule-List and Privacy Policy) A rule-list for a vocabulary Voc is a list containing elements of $U \times D \times P \times A \times \{+, \circ, -\} \times C(Var) \times \mathfrak{P}(O)$. For ease of handling, we write a rule $(u, d, p, a, r, c, \bar{o})$ as $\langle (u, d, p, a), (r, c, \bar{o}) \rangle$ and we call (u, d, p, a) the scope and (r, c, \bar{o}) the qualifier of the rule.

A privacy policy or EPAL policy is a tuple $(Voc, R, gc, dr, \bar{do})$ of a vocabulary Voc , a rule-list R for Voc , a global condition $gc \in C(Var)$, a default ruling $dr \in \{+, \circ, -\}$, and a default obligation $\bar{do} \in \mathfrak{P}(O)$. The set of these policies is called EPAL, and the subset for a given vocabulary $EPAL(Voc)$. \diamond

In EPAL, precedences are contained implicitly by the textual order of the rules. The rulings $+$, \circ , and $-$ mean ‘allow’, ‘don’t care’, and ‘deny’. The ruling \circ was not yet present in [3]. In EPAL, it is called ‘obligate’ because it enables rules that do not make a decision but only impose additional obligations. An example is the rule “Whenever someone tries to access my data, I want to receive a notification”.

As a naming convention, we assume that the components of a privacy policy called Pol are always called as in Definition 2.6, and if Pol has a sub- or superscript, then so do the components.

2.3 Semantics of EPAL Policies

A request is a tuple (u, d, p, a) , which should belong to the set $U \times D \times P \times A$ for the given vocabulary. Note that EPAL requests are not restricted to “ground terms” as in some other languages, i.e., minimal elements in the hierarchies. This is useful if one starts with coarse policies and refines them because elements that are initially minimal may later get children. For instance, the individual users in a “department” of an “enterprise” may not be mentioned in the CPO’s privacy policy, but in the department privacy policy. For similar reasons, the semantics is also defined for requests outside the given vocabulary.

Definition 2.7 (Request) For a vocabulary Voc , we define the set of valid requests as $Req(Voc) := U \times D \times P \times A$. \diamond

Whether a rule with a satisfied condition matches a given request depends on its ruling. We say that a rule is negative if it has a ‘deny’ ruling, otherwise it is positive. A positive rule matches for a parent of the request (in all hierarchies) including the request itself, i.e., these rules are inherited downward the hierarchies. If the ruling is ‘deny’, then the rule also matches if it is specified for a child of the request, i.e., these rules are additionally inherited upward the hierarchies. The reason is that the hierarchies are considered groupings; if access is forbidden to an element of a group, it is also forbidden for the group as a whole.

Definition 2.8 (Matching rule) Let $(u, d, p, a) \square (u', d', p', a')$ iff $u \square u' \wedge d \square d' \wedge p \square p' \wedge a \square a'$ for $\square \in \{\geq, \geq\}$. A positive (negative) rule $\langle (u, d, p, a), (r, c, \bar{o}) \rangle$ matches a request (u', d', p', a') iff $(u, d, p, a) \geq (u', d', p', a')$ ($(u, d, p, a) \geq (u', d', p', a')$). \diamond

The semantics of a privacy policy Pol is a function $eval_{Pol}$, given in Algorithm 1, that evaluates a request based on a given assignment and returns the result (r, \bar{o}) of a ruling (decision) and associated obligations.

```

Input: A policy  $P = (Voc, R, gc, dr, \bar{d}o)$ , request  $req = (u_R, d_R, p_R, a_R)$  and assignment  $\chi \in \mathcal{Ass}(Var)$ 
Output:  $eval_{Pol}(P, req) \in \{(scope\_error, \emptyset), (policy\_error, \emptyset)\} \cup \{+, \circ, -\} \times O$ 
if  $(u_R, d_R, p_R, a_R) \notin U \times D \times P \times A$  then return  $(scope\_error, \emptyset)$ 
if  $eval_\chi(gc) = false$  then return  $(policy\_error, \emptyset)$ 
 $\bar{o}_{add} := \emptyset$ 
foreach  $\langle (u, d, p, a), (r, c, \bar{o}) \rangle \in R$  do
  if  $eval_\chi(c) = true$  then
    if  $r = + \wedge (u, d, p, a) \geq (u_R, d_R, p_R, a_R)$  then return  $(r, \bar{o} \cup \bar{o}_{add})$ 
    if  $r = - \wedge (u, d, p, a) \geq (u_R, d_R, p_R, a_R)$  then return  $(r, \bar{o} \cup \bar{o}_{add})$ 
    if  $r = \circ \wedge (u, d, p, a) \geq (u_R, d_R, p_R, a_R)$  then  $\bar{o}_{add} = \bar{o}_{add} \cup \bar{o}$ 
return  $(dr, \bar{o}_{add} \cup \bar{d}o)$ 

```

Algorithm 1: Request Evaluation

If the request is not valid for the considered vocabulary or the global condition is satisfied under the given assignment then the result is $(scope_error, \emptyset)$ or $(policy_error, \emptyset)$, respectively. Otherwise, the output ruling is determined by the first matching rule with ‘allow’ or ‘deny’ ruling and whose condition is satisfied. If no such rule exists, the default ruling applies. The obligations of preceding obligate rules are added to the result.

2.4 Refinement of Privacy Policies

We finally review the notion of refinement for EPAL policies, which is the foundation of almost all operations on policies.

Our notion of refinement allows policy Pol_2 to define a ruling if Pol_1 does not care. Additionally, it is allowed to extend the scope of the original policy and to define arbitrary rules for the new elements. In all other cases, the rulings of both policies must be identical. This also comprises the ruling *conflict_error*. For new elements however, we have to capture that if they are appended to the existing hierarchies, there could exist applicable rules for these elements if they were already present, and newly added rules for these elements could influence existing elements as well. As an example, a rule for a “department” may forbid its “employees” to access certain data for marketing purposes. Now if a new employee is added, this rule should as well be applicable; furthermore, defining a new rule for this case with higher precedence, e.g., granting the new employee an exception to the department’s rule should obviously not yield a refinement any more. In our definition of refinement, we therefore do not evaluate each policy on its own vocabulary but on the joint vocabulary of both policies. One technicality that has to be taken care of is that joining two vocabularies, i.e., joining their respective hierarchies, might not yield another vocabulary. Hence, we only define refinement for policies with compatible vocabularies, i.e., those policies for which joining their respective vocabularies pair-wise gives another hierarchy again.

Dealing with the respective obligations is somewhat more difficult. Intuitively, one wants to express that a finer policy may also contain refined obligations. However, since a refined policy might contain additional obligations, whereas some others have been omitted, it is not possible to simply compare these obligations in the obligation model of the original policy. (Recall that we also use refinement to compare arbitrary policies; hence one cannot simply expect that all vocabulary parts of the refined policy are supersets of those of the coarser policy.) The following notion of obligation refinement is from [4].

Definition 2.9 (Obligation Refinement) *Let two obligation models (O_i, \rightarrow_{O_i}) and $\bar{o}_i \subseteq O_i$ for $i = 1, 2$ be given. Then \bar{o}_2 is a refinement of \bar{o}_1 , written $\bar{o}_2 \prec \bar{o}_1$, iff the following holds:*

$$\exists \bar{o} \subseteq O_1 \cap O_2: \bar{o}_2 \rightarrow_{O_2} \bar{o} \rightarrow_{O_1} \bar{o}_1.$$

◇

We are now ready to introduce our notion of policy refinement.

Definition 2.10 (Policy Refinement) *Let two privacy policies $Pol_i = (Voc_i, R_i, gc_i, dr_i, \bar{d}o_i)$ for $i = 1, 2$ with compatible vocabularies be given, and set $Pol_i^* = (Voc_i^*, R_i, gc_i, dr_i, \bar{d}o_i)$ for $i = 1, 2$, where $Voc_i^* = (UH_1 \cup$*

$UH_2, DH_1 \cup DH_2, PH_1 \cup PH_2, AH_1 \cup AH_2, Var_i, OM_i$). Then Pol_2 is a refinement of Pol_1 , written $Pol_2 \prec Pol_1$, iff for every assignment $\chi \in \mathcal{Ass}(Var_1 \cup Var_2)$ and every authorization request $q \in Req$ one of the following statements holds, where $(r_i, \bar{o}_i) = eval_{Pol_i}(q, \chi)$ for $i = 1, 2$:

- $(r_1, \bar{o}_1) = (scope_error, \emptyset)$.
- If $eval_\chi(gc_1) = false$ then also $eval_\chi(gc_2) = false$.
- $r_1 \in \{+, -\}$ and $r_2 = r_1$ and $\bar{o}_2 \prec \bar{o}_1$.
- $r_1 = \circ$ and $r_2 \in \{+, \circ, -\}$ and $\bar{o}_2 \prec \bar{o}_1$.

◇

The trivial solution for implementing policy refinement is the brute force approach, i.e., one simply evaluates both policies for any request and any assignment, and compare the results. Clearly, a brute force search is not desirable, and we can identify three inherent weaknesses of this approach that we will address in our algorithm.

First, the processing is performed for all element of the joint set of valid requests. In case several requests have exactly the same matching rules in the rule-list, it would be beneficial to group together these quadruples and perform a single processing for all of them.

Secondly, in order to cover all the combinations of conditions which could be satisfied by a given request, the brute force algorithm has to consider all the possible subsets of the sets of conditions defined in the two compared policies. However, it might be that some subsets do not have to be considered since several conditions cannot be satisfied at the same time. It could hence be beneficial to restructure the set of rules with respect to their conditions.

Finally, several rules will typically useless for a particular request and assignment since they are always hidden by matching rules which have higher priority. One should hence restructure the rule list in a suitable way.

We will address these weaknesses in the next section by our *scope-based approach* for comparing privacy policies.

3 Scope-based Policy Comparison

This section describes our algorithm for policy refinement, called *scope-based policy comparison*. The algorithm consists of four parts:

1. The *scope-based expansion* transforms the rule-list of a policy into an ordered list of so-called scope-based rules. In contrast to usual rules, scope-based rules consist of a sequence of qualifiers instead of a single qualifier. The derived list of rules is equivalent to the old one in the sense that the evaluation of each request results in the same output for each possible assignment. However, the derived list enjoys a property that is crucial for the correctness of the following phases, namely that rules that are matching for only a small number of elements come first.
2. Given two such policies with ordered lists of scope-based rules, we *normalize* the qualifier sequences of each rule according to a simple calculus. The essential ideas are to eliminate qualifiers with obligate ruling by accumulating the respective obligations and to close the sequence, if necessary, with the qualifier $(dr, true, d\bar{o})$.
3. After the two previous parts, which are used for pre-processing policies, we now show how to efficiently check if two normalized qualifier sequences are refining in the sense that for every assignment, both sequences yield the same output and one policy always yields refined obligations.
4. We finally show how to efficiently check for refinement between two policies that have scope-based rule-lists with normalized qualifier sequences.

In the following, we decided not to present a precise description of the algorithm including all the tedious details that it has to take care of, both for reasons of readability and for space constraints, but we illustrate its different parts by means of examples instead. However, the precise definition of the algorithm can easily be derived from our description.

3.1 Scope-based Expansion

An important prerequisite for scope-based rules is the notion of *extended rules*. Instead of having only one qualifier, an extended rule may have a sequence of qualifiers. For example, a rule $\langle(u, d, p, a), (r_1, c_1, \bar{o}_1)\rangle$ followed by another rule $\langle(u, d, p, a), (r_2, c_2, \bar{o}_2)\rangle$ can be described by the extended rule $\langle(u, d, p, a), \langle(r_1, c_1, \bar{o}_1); (r_2, c_2, \bar{o}_2)\rangle\rangle$, where evaluation of qualifiers is from left to right and thus respects the precedences of the original rules.

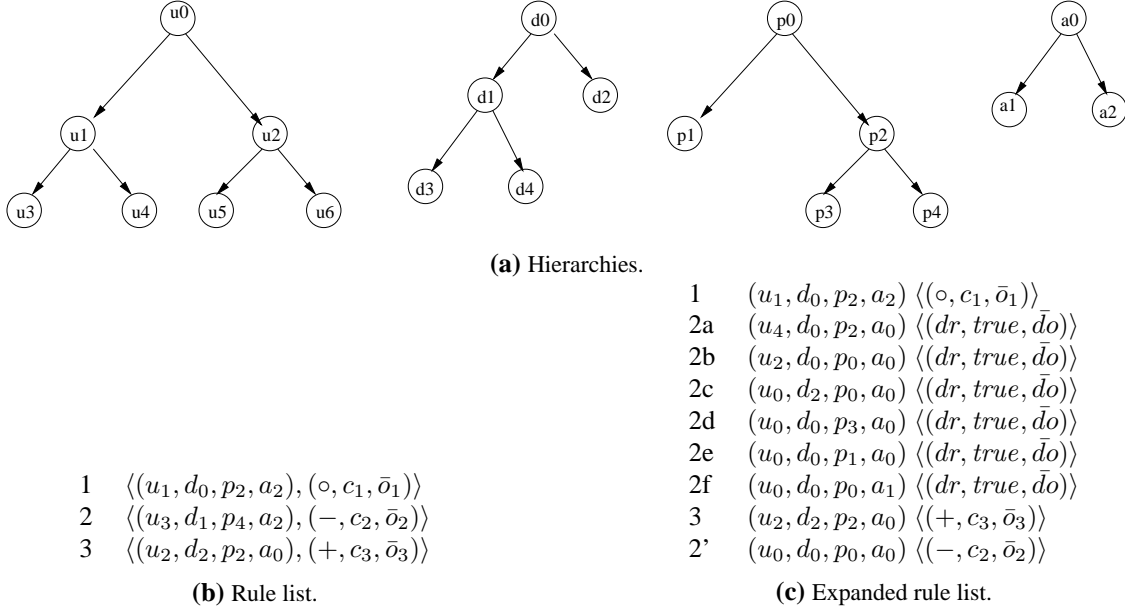


Figure 1: Example policy. Dashed circles indicate the scopes of the rules in the respective dimensions.

Note that for positive rules, all elements affected by such a rule can easily be represented by their parent element, i.e., the element that the rule is defined for. In contrast, deny rules do not have such a compact representation since upward inheritance prevents us from describing all affected elements by means of a single element. However, we can describe a deny rule for an element (u, d, p, a) by a deny rule for the whole hierarchies, i.e., a rule for the root element, but explicitly excluding the siblings on the path to the root as defined below (for ease of description, we assume that each hierarchy has only a single root, which we denote as (u_0, d_0, p_0, a_0)):

$$\begin{aligned}
 & \text{siblings}(\langle(u, d, p, a), \text{seq}\rangle) \\
 &= \{ \langle(u', d_0, p_0, a_0), \langle(dr, \text{true}, \bar{d}o)\rangle \mid u_0 > u' > u \} \cup \{ \langle(u_0, d', p_0, a_0), \langle(dr, \text{true}, \bar{d}o)\rangle \mid d_0 > d' > d \} \\
 & \cup \{ \langle(u_0, d_0, p', a_0), \langle(dr, \text{true}, \bar{d}o)\rangle \mid p_0 > p' > p \} \cup \{ \langle(u_0, d_0, p_0, a'), \langle(dr, \text{true}, \bar{d}o)\rangle \mid a_0 > a' > a \}
 \end{aligned}$$

Roughly, this alternative representation and re-shuffling of the rules allows us to generate a “normal form” for rule-lists. Although the normal form has more rules than the original rule-list, it simplifies the comparison of the rule-lists since each individual rule does not necessarily have to be compared with all rules of the other rule-list.

We describe the transformation from the original rule-list to this normal form and to the final list of scope-based rules by means of an example policy, whose hierarchies and rule-list are depicted in Fig. 1a and Fig. 1b, respectively. The first step towards the scope-based rule-list is to switch to the described representation of all deny rules (i.e., of rule 2 in the example). This means that we first extend each deny rule to all elements of the hierarchies and then explicitly exclude those elements that must not be affected by this artificially enlarged scope. Formally, this corresponds to a new rule for the root and additional rules for the respective siblings. However, we have to ensure that the remaining allow rules (i.e., rule 3) is not affected by the artificially inserted deny rule that covers all elements. Formally, this means that we have to shift the allow rule before the global deny rule. This is shown in Fig. 1c.

Next, we let all original obligation rules float down the rule-list as follows. We have to distinguish four cases:

1. If there is no overlap with the next lower rule, i.e., there are no elements for which both rules are matching, we swap both rules (as done in Steps (ii) & (v) in Fig. 2).

2. If the scope of the floating rule is contained in the scope of the next rule, the qualifier of that rule is appended to the floating rule's qualifier and the processing of this obligation rule is finished. == stop rearranging.
3. If the scope of the next rule is contained in the scope of the floating rule, we swap both rules but additionally append the qualifier of the floating rule to the qualifier sequence of the current rule.
4. Finally, if both rules only overlap partially, we swap the rules and additionally insert a new rule that deals with the overlap as follows:

$$\text{overlap}(\langle (u, d, p, a) \text{ seq}_1 \rangle, \langle (u', d', p', a') \text{ seq}_2 \rangle) =: \langle (u^*, d^*, p^*, a^*) \text{ seq}_1 \rangle$$

where $u^* = \begin{cases} u & \text{if } u \leq_U u' \\ u' & \text{otherwise} \end{cases}$, and similarly for the other dimensions. This is shown in Steps (i), (iii), and (iv) in Fig. 2.

After we have processed all obligation rules in this way, we let the positive rules float up until we reach a rule whose scope comprises the scope of the allow rule. In the example policy, rule 3 in Fig. 3a floats up to the top as there are only either non-overlapping rules (2e, 2d, 2a, 2a') or partially overlapping obligation rules. This finally yields the desired scope-based rule-list. The following lemmas capture the important properties of scope-based rule-lists:

Lemma 3.1 *Let $P = (\text{Voc}, R, gc, dr, \bar{d}o)$ be a privacy policy and let SR denote the scope-based rule-list of R . Let $\sigma = \langle (u, d, p, a), \text{seq} \rangle$ and $\sigma' = \langle (u', d', p', a'), \text{seq}' \rangle$ be arbitrary rules in SR . If $\text{scope}(u, d, p, a) \subset \text{scope}(u', d', p', a')$ then σ has higher precedence than σ' . \square*

Lemma 3.2 *Let $P = (\text{Voc}, R, gc, dr, \bar{d}o)$ be a privacy policy and let SR denote the scope-based rule-list of R . Then for every valid request (u_R, d_R, p_R, a_R) for which there exists a matching rule in R , the following holds:*

- *There exists a rule in SR that matches for (u_R, d_R, p_R, a_R) .*
- *Let $\langle (u, d, p, a), \text{seq} \rangle$ denote the rule with the highest precedence in SR and let (u_R, d_R, p_R, a_R) be an arbitrary element in the scope of (u, d, p, a) . Then seq contains the qualifiers from all matching rules in R for (u_R, d_R, p_R, a_R) .*

\square

Before we continue with the next part of the algorithm, we additionally want to cover those requests for which there is no matching rule in the rule-list; in particular, we have to consider those requests that are valid requests for the policy that we want to compare our policy with. Thus, for every root (u^*, d^*, p^*, a^*) of the combined vocabularies for which there does not already exist a matching rule, the rule $\langle (u^*, d^*, p^*, a^*), (dr, \text{true}, \bar{d}o) \rangle$ is appended to the rule-list.

3.2 Normalization of Qualifier Sequences

In this part, qualifier sequences are transformed into equivalent, so-called *normalized* sequences, that do not contain qualifiers with obligate ruling anymore and that each sequence ends with qualifier $(dr, \text{true}, \bar{d}o)$. We will see in Section 3.3 that two qualifier sequences of this special form can easily be compared. We describe the transformation by the following 9 axioms, which are used as rewriting rules.

The axioms (1) and (2) state that an obligate ruling can always be shifted to the right by a suitable adoption of conditions and obligations. Note that there is no axiom for two subsequent qualifiers with obligate ruling.

$$\frac{\langle o, c_1, \bar{o}_1 \rangle; \langle r, c_2, \bar{o}_2 \rangle}{\langle r, c_1, \bar{o}_1 \& \bar{o}_2 \rangle} \quad c_1 \Rightarrow c_2, r \in \{+, -\} \quad (1)$$

$$\frac{\langle o, c_1, \bar{o}_1 \rangle; \langle r, c_2, \bar{o}_2 \rangle}{\langle r, c_1 \wedge c_2, \bar{o}_1 \& \bar{o}_2 \rangle} \quad \langle r, c_2, \bar{o}_2 \rangle \quad \langle o, c_1, \bar{o}_1 \rangle \quad \neg(c_1 \Rightarrow c_2), r \in \{+, -\} \quad (2)$$

The axioms (3) and (4) are used to simplify a qualifier sequence. They omit qualifiers, which are “hidden” beyond a qualifier with higher precedence, and derive a more useful representation of conditions.

$$\frac{\langle r_1, c_1, \bar{o}_1 \rangle; \langle r_2, c_2, \bar{o}_2 \rangle}{\langle r_1, c_1, \bar{o}_1 \rangle} \quad c_2 \Rightarrow c_1, r_1 \in \{+, -\}, r_2 \in \{+, -, \circ\} \quad (3)$$

$$\frac{\langle r_1, c_1, \bar{o}_1 \rangle; \langle r_2, c_2, \bar{o}_2 \rangle}{\langle r_1, c_1, \bar{o}_1 \rangle; \langle r_2, c_2 \wedge \neg c_1, \bar{o}_2 \rangle} \quad \neg(c_2 \Rightarrow c_1), r_1 \in \{+, -\}, r_2 \in \{+, -, \circ\} \quad (4)$$

In case the transformed sequence generated by the application of the already described axioms does not end with the qualifier $\langle dr, true, \bar{d}\bar{o} \rangle$, we want to be able to explicitly append this qualifier to the qualifier sequence. This is captured in axiom 5.

$$\frac{\langle o, c, \bar{o} \rangle.}{\langle dr, c, \bar{o} \& \bar{d}\bar{o} \rangle; \langle dr, true, \bar{d}\bar{o} \rangle} \quad (5)$$

As an example, consider the qualifier sequence (6) that we want to transform into normal form. Applying the axioms 2, 1, 1, and 3, we get the rearranged sequence (7), where hidden qualifiers are removed and obligations with obligate ruling are pushed into qualifiers with allow or deny ruling. The sequence is terminated by an “otherwise” qualifier, which returns the default ruling and default obligation of the policy.

$$\langle o, c_1 \wedge c_2, \bar{o}_7 \rangle; \langle +, c_1, \bar{o}_6 \rangle; \langle -, c_1 \wedge c_2, \bar{o}_5 \rangle; \langle o, c_2, \bar{o}_4 \rangle; \langle -, c_2, \bar{o}_1 \rangle \quad (6)$$

$$\langle +, c_1 \wedge c_2, \bar{o}_6 \& \bar{o}_7 \rangle; \langle -, c_2, \bar{o}_4 \& \bar{o}_1 \rangle; \langle +, c_1, \bar{o}_6 \rangle; \langle dr, true, \bar{d}\bar{o} \rangle \quad (7)$$

For optimization, we finally also introduce the following two axioms. They correspond to elimination rules that remove qualifiers which will not be applied:

$$\frac{\langle r_1, c_1, \bar{o}_1 \rangle; \langle r_2, c_2, \bar{o}_2 \rangle}{\langle r_1, c_1, \bar{o}_1 \rangle} \quad c_2 \Rightarrow false, r_1 \in \{+, -\}, r_2 \in \{+, -, \circ\} \quad (8)$$

$$\frac{\langle r_1, c_1, \bar{o}_1 \rangle; \langle r_2, c_2, \bar{o}_2 \rangle}{\langle r_2, c_2, \bar{o}_2 \rangle} \quad c_1 \Rightarrow false, r_1 \in \{+, -\}, r_2 \in \{+, -, \circ\} \quad (9)$$

The following lemma summarizes the main property of normalized qualifier sequences.

Lemma 3.3 *Let (r, c, \bar{o}) and (r', c', \bar{o}') be two qualifiers in a normalized sequence seq , and let \Rightarrow be a correct implies relation for the considered vocabulary. Then the following holds:*

1. *If $c \Rightarrow c'$ then (r, c, \bar{o}) has higher precedence than (r', c', \bar{o}') , i.e., it comes first in the sequence.*
2. *For any assignment χ for the considered vocabulary, there exists at least one qualifier in seq whose condition is satisfied under χ .*

□

3.3 Comparison of Qualifier Sequences

The comparison of two sequences checks whether there is a refinement for every possible pair of qualifiers. Condition comparison is based on the considered implies relation, which we assume to be correct. To illustrate the comparison process, we consider the normalized qualifier sequences (10) and (11).

$$\langle r_1, c_1 \wedge c_2 \wedge c_3, \bar{o}_1 \rangle; \langle r_2, c_1 \wedge c_2, \bar{o}_2 \rangle; \langle r_3, c_2, \bar{o}_3 \rangle; \langle r_4, c_1, \bar{o}_4 \rangle; \langle dr_1, true, \bar{d}\bar{o}_1 \rangle \quad (10)$$

$$\langle r'_1, c_1 \wedge c_3, \bar{o}'_1 \rangle; \langle r'_2, c_3, \bar{o}'_2 \rangle; \langle r'_3, c_1, \bar{o}'_3 \rangle; \langle dr_2, true, \bar{d}\bar{o}_2 \rangle \quad (11)$$

Roughly, for each qualifier in sequence (11) and in descending order, we check those qualifiers in sequence (10) for refinement whose conditions could be concurrently satisfied until we reach a qualifier whose condition implies the qualifier’s condition of sequence (11). If we obtain a refinement for this qualifier, we will explain this in more detail below, we proceed with the next qualifier of sequence (11), until we finally reach a qualifier in the sequence (11) whose

condition must be fulfilled under the assumption that the condition of the currently investigated qualifier of sequence (10) holds. After this refinement check is also successful, we proceed with the next element of sequence (10).

In the example, we start with the qualifier $(r'_1, c_1 \wedge c_3, \bar{o}'_1)$ and we assume that the condition $c_1 \wedge c_3$ is true. We process the elements of sequence (10) in descending order until a qualifier with satisfied condition is found. Since the condition $c_1 \wedge c_2 \wedge c_3$ of the first qualifier $(r_1, c_1 \wedge c_2 \wedge c_3, \bar{o}_1)$ may be true concurrently, we have to compare (r_1, \bar{o}_1) and (r'_1, \bar{o}'_1) . More precisely, we have to check that if $r_1 \neq \circ$ we have $r'_1 = r_1$; moreover \bar{o}'_1 must refine \bar{o}_1 . If this holds we continue the comparison with the next qualifier in sequence (10).

We know at this point that $((c_1 \wedge c_3) \wedge \neg(c_1 \wedge c_2 \wedge c_3))$ holds. Since the implies relation is correct, we obtain $((c_1 \wedge c_3) \wedge \neg(c_1 \wedge c_2 \wedge c_3)) \Rightarrow \neg(c_1 \wedge c_2)$ hence the condition $(c_1 \wedge c_2)$ cannot be true. This means that the qualifier $(r_2, c_1 \wedge c_2, \bar{o}_2)$ does not have to be considered. The same holds for qualifier (r_3, c_2, \bar{o}_3) . Because of $c_1 \wedge c_3 \Rightarrow c_1$, (r_4, c_1, \bar{o}_4) is the next matching qualifier and the tuples (r_4, \bar{o}_4) and (r'_1, \bar{o}'_1) have to be compared. Moreover, since $c_1 \wedge c_3$ implies c_1 , no remaining elements in sequence (10) must be checked.

We continue the comparison with the second qualifier in sequence (11). At this point we know that c_3 and $\neg(c_1 \wedge c_3)$ hold. Because condition $c_1 \wedge c_3$ does not hold, the qualifier $(r_1, c_1 \wedge c_2 \wedge c_3, \bar{o}_1)$ cannot apply. Further, because of $\neg(c_1 \wedge c_2) \wedge c_3$, the same holds for the qualifiers $(r_2, c_1 \wedge c_2, \bar{o}_2)$, (r_3, c_2, \bar{o}_3) , and (r_4, c_1, \bar{o}_4) . Thus, we have to check (r'_2, c_3, \bar{o}'_2) with $(dr_1, true, d\bar{o}_1)$.

Similarly, we check the remaining elements in sequence (11). The processing of all qualifiers in sequence (11) is summarized in Table 1.

Satisfied Condition	Result given by seq (10)	Result given by seq (11)
$c_1 \wedge c_2 \wedge c_3$	(r_1, \bar{o}_1)	(r'_1, \bar{o}'_1)
$c_1 \wedge c_3$	(r_4, \bar{o}_4)	(r'_1, \bar{o}'_1)
$c_2 \wedge c_3$	(r_3, \bar{o}_3)	(r'_2, \bar{o}'_2)
c_3	$(dr_1, d\bar{o}_1)$	(r'_2, \bar{o}'_2)
$c_1 \wedge c_2$	(r_2, \bar{o}_2)	(r'_3, \bar{o}'_3)
c_1	(r_4, \bar{o}_4)	(r'_3, \bar{o}'_3)
c_2	(r_3, \bar{o}_3)	$(dr_2, d\bar{o}_2)$
--	$(dr_1, d\bar{o}_1)$	$(dr_2, d\bar{o}_2)$

Table 1: Request Evaluation Results Comparison

Lemma 3.4 *Do we need a lemma here, similar to the previous sections?* □

3.4 Comparison of extended rule-lists

Finally, we are ready to check for refinement of two privacy policies by comparing their normalized, scope-based rule-lists. If there is refinement for the qualifier sequences of all “matching” rules then there is policy refinement.

Let SR_i for $i = 1, 2$ denote two scope-based rule-lists. Let $\sigma_2 = \langle (u_2, d_2, p_2, a_2), seq_2 \rangle$ be a rule in SR_2 . Processing SR_1 in descending precedence, we check each overlapping rule $\sigma_1 = \langle (u_1, d_1, p_1, a_1), seq_1 \rangle$ whether the qualifier sequences seq_2 and seq_1 constitute a refinement. If there is no refinement, the algorithm stops and returns *false*. The processing finishes when a rule σ'_1 with $scope(\sigma'_1) \subseteq scope(\sigma_2)$ is found. This is always the case because every SR ends with rule(s) covering the whole hierarchies (by construction).

To illustrate the comparison, consider the two scope-based rule-lists depicted in Fig. 4. The goal of the comparison is to test whether every request evaluation result in SR_2 refines the corresponding evaluation result in SR_1 . Thus, for every rule in SR_2 , all possible matching rules in SR_1 are tested.

<ol style="list-style-type: none"> 1 $(u_2, d_2, p_2, a_0) seq_1$ 2 $(u_4, d_0, p_2, a_2) seq_2$ 3 $(u_4, d_0, p_0, a_0) seq_3$ 4 $(u_1, d_0, p_2, a_2) seq_4$ 5 $(u_0, d_0, p_0, a_0) seq_5$ <p style="text-align: center;">List SR_1</p>	<ol style="list-style-type: none"> 1' $(u_4, d_0, p_0, a_0) seq'_1$ 2' $(u_0, d_0, p_0, a_0) seq'_2$ <p style="text-align: center;">List SR_2</p>
---	--

Figure 4: Example extended rule list comparison.

In descending order, we check each rule in SR_2 with rules in SR_1 whose scopes overlap, comparing their qualifier sequences as described in Section 3.3. Thus, we begin with rule 1'. The first overlap is with rule 2 ($scope(u_4, d_0, p_2, a_2) \subseteq scope(u_4, d_0, p_0, a_0)$). If the qualifier sequences seq'_1 and seq_2 are a refinement then we continue else SR_2 is not a refinement of SR_1 and we stop. The next overlapping rule is rule 3. After successful comparison we do not have to check the remaining rules in SR_1 as the scope of rule 3 completely covers the scope of rule 1'. We continue with the second rule in SR_2 and check overlap with the rules in SR_1 in descending order. Because rule 2' has scope (u_0, d_0, p_0, a_0) , the qualifier sequence of every rule in SR_1 must be checked.

The structure of SR_i allows for testing the refinement for all the possible valid requests. Thus, if the refinement is verified for all the compared sequences of qualifiers then the comparison algorithm returns *true*.

Theorem 3.1 *Let $P_i = (V_i, R_i, gc_i, dr_i)$ for $i = 1, 2$ be two privacy policies. Let further P_i^* for $i = 1, 2$ denote the policies that are derived as in Definition 2.10, and let a correct implies relation for the considered vocabularies be given. Then the following holds:*

If the scope-based comparison algorithm applied on P_1^ and P_2^* outputs true, then P_1 is a refinement of P_2 . Moreover, if the implies relation is additionally complete, then the converse direction also holds, i.e., if P_1 is a refinement of P_2 then the scope-based comparison algorithm outputs true. \square*

4 Conclusion

References

- [1] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise Privacy Authorization Language (EPAL). Research Report 3485, IBM Research, 2003. <http://www.zurich.ibm.com/security/enterprise-privacy/epal/specificatio%n>.
- [2] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise Privacy Authorization Language (EPAL). Research Report RZ 3485, IBM Research, Mar. 2003.
- [3] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. E-P3P privacy policies and privacy authorization. In *Proc. 1st ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 103–109, 2002.
- [4] M. Backes, B. Pfitzmann, and M. Schunter. A toolkit for managing enterprise privacy policies. In *European Symposium on Research in Computer Security (ESORICS)*, Lecture Notes in Computer Science, pages 101–119. Springer-Verlag, Berlin Germany, 2003.
- [5] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekerat. Obligation monitoring in policy management. In *Proc. 3rd IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 2–12, 2002.
- [6] P. A. Bonatti, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. A component-based architecture for secure data publication. In *Proc. 17th Annual Computer Security Applications Conference*, pages 309–318, 2001.
- [7] A. Cavoukian and T. J. Hamilton. *The Privacy Payoff: How successful businesses build customer trust*. McGraw-Hill/Ryerson, 2002.
- [8] S. Fischer-Hübner. *IT-security and privacy: Design and use of privacy-enhancing security mechanisms*, volume 1958 of *Lecture Notes in Computer Science*. Springer, 2002.
- [9] S. Jajodia, M. Kudo, and V. S. Subrahmanian. Provisional authorization. In *Proc. E-commerce Security and Privacy*, pages 133–159. Kluwer Academic Publishers, 2001.
- [10] G. Karjoth and M. Schunter. A privacy policy model for enterprises. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 271–281, 2002.
- [11] G. Karjoth, M. Schunter, and M. Waidner. The platform for enterprise privacy practices – privacy-enabled management of customer data. In *Proc. Privacy Enhancing Technologies Conference*, volume 2482 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2002.

- [12] Platform for Privacy Preferences (P3P). W3C Recommendation, Apr. 2002. <http://www.w3.org/TR/2002/REC-P3P-20020416/>.
- [13] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies with complex constraints. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2001.
- [14] TRUSTe. Privacy Certification. Available at www.truste.com.
- [15] eXtensible Access Control Markup Language (XACML). OASIS Committee Specification 1.0, Dec. 2002. www.oasis-open.org/committees/xacml.