

A REVIEW OF TRUST MANAGEMENT, SECURITY AND PRIVACY POLICY LANGUAGES

Juri Luca De Coi, Daniel Olmedilla

*L3S Research Center, University of Hannover, Appelstr. 9a, D-30167 Hannover, Germany
decoi@L3S.de, olmedilla@L3S.de*

Keywords: Policy languages, comparison

Abstract: Policies are a well-known approach to protecting security and privacy of users as well as for flexible trust management in distributed environments. In the last years a number of policy languages were proposed to address different application scenarios. In order to help both developers and users in choosing the language best suiting her needs, policy language comparisons were proposed in the literature. Nevertheless available comparisons address only a small number of languages, are either out-of-date or too narrow in order to provide a broader picture of the research field. In this paper we consider twelve relevant policy languages and compare them on the strength of ten criteria which should be taken into account in designing every policy language. Some criteria are already known in the literature, others are introduced in our work for the first time. By comparing the choices designers made in addressing such criteria, useful conclusions can be drawn about strong points and weaknesses of each policy language.

1 Introduction

Security management is a foremost issue in large scale networks like the World Wide Web. In such a scenario, traditional assumptions for establishing and enforcing access control regulations do not hold anymore. In particular identity-based access control mechanisms have proved to be ineffective, since in decentralized and multicentric environments, the requester and the service provider are often unknown to each other.

Policies are a well-known approach to protecting security and privacy of users in the context of the Semantic Web: policies specify who is allowed to perform which action on which object depending on properties of the requester and of the object as well as parameters of the action and environmental factors (e.g., time).

The potential policies have proved to own is not fully exploited yet, since nowadays their usage is mainly restricted to specific application areas. Lacking knowledge about currently available solutions is one of the main factors hindering widespread use of policies: in order to exploit a policy language the po-

tential user needs to be provided with a clear picture of the advantages it provides in comparison with other solutions. Furthermore in the last years many policy languages were proposed, targeting different application scenarios and provided with different features and expressiveness: scope and properties of available languages have to be known to the user in order to help her in choosing the one most suitable to her needs.

In an attempt to help with these and other problems, comparisons among policy languages have been provided in the literature, anyway existing comparisons either do not consider a relevant number of available solutions or are mainly focused on the application scenarios the authors worked with (e.g., trust negotiation in (Seamons et al., 2002) or ontology-based systems in (Tonti et al., 2003)), moreover policy-based security management is a rapidly evolving field and most of this comparison work is now out-of-date.

Currently a broad and up-to-date overview covering most of the relevant available policy languages is lacking; in this paper we intend to fill this gap by providing an extensive comparison covering twelve pol-

icy languages. Such a comparison will be carried out on the strength of ten criteria, partly already known in the literature and partly introduced in our work for the first time. Our analysis will hopefully help users in choosing the policy language mostly suiting their needs, as well as researchers currently investigating this area.

This paper is organized as follows. In section 2 related work is accounted for. Section 3 briefly sketches the evolution of the research field and introduces some concepts (e.g., role-based policy language as well as various kinds of policies) which will be massively exploited in the following. Sections 4 and 5 respectively introduce the languages which will be compared in the rest of the paper and the criteria according to which the comparison will be carried out. The actual comparison takes place in section 6, whereas sections 7 and 8 respectively present overall results and draw some conclusions.

2 Related work

The paper of Seamons et al. (Seamons et al., 2002) is the basis of our work: some of the insights they suggested have proved to be still valuable right now and as such they are addressed in our work as well. Nevertheless in over three years the research field has considerably changed and nowadays many aspects of (Seamons et al., 2002) are out of date: new languages have been developed and new design paradigms have been taken into account, what makes the comparison performed in (Seamons et al., 2002) obsolete and many criteria according to which they were evaluated not suitable anymore.

The pioneer paper of Seamons et al. paved the way to future research on policy language comparisons like Tonti et al. (Tonti et al., 2003), Anderson (Anderson,) and Duma et al. (Duma et al.,): although (Tonti et al., 2003) actually presents a comparison of two ontology-based languages (namely KAoS and Rei) with the object-oriented language Ponder, the work is rather an argument for ontology-based systems, since it clearly shows the advantages of ontologies.

Because of the impressive amount of details it provides, (Anderson,) restricts the comparison to only two (privacy) policy languages, namely EPAL and XACML, therefore a comprehensive overview of the research field is not provided, and features which neither EPAL nor XACML support are not taken into account at all among the comparison criteria.

Finally (Duma et al.,) provides a comparison specifically targeted to giving insights and sugges-

tions to policy writers (*designers*): therefore the criteria, according to which the comparison is carried out, are mainly practical ones and scenario-oriented, whereas more abstract issues are considered out of scope and hence not addressed.

3 Background

In this section some concepts are introduced, which will help to smoothly understand the rest of the paper. First an overall picture of the research field is provided by briefly outlining the historical evolution of policy languages, then the definitions of some policy types which will be used throughout the paper are provided.

3.1 From uid/psw-based authentication to trust negotiation

Traditional access control mechanisms (like the ones exploited in traditional operating systems) make authorization decisions based on the identity of the requester: the user must provide a pair (*username, password*) and, if this pair matches with one of the entry in some static table kept by the system (e.g., the file `/etc/passwd` in Unix) the user is granted with some privileges. However, in decentralized or multicentric environments, peers are often unknown to each other, and access control based on identities may be ineffective. In order to address this scenario, role-based access control mechanisms were developed. In a role-based access control system a user is assigned with one or more roles, which are in turn exploited in order to take authorization decisions. Since the number of roles is typically much smaller than the number of users, role-based access control systems reduce the number of access control decisions. A thorough description of role-based access control can be found in (Herzberg et al., 2000).

In a role-based access control system the authorization process is split into two steps, namely assignment of one or more roles and check whether a member of the assigned role(s) is allowed to perform the requested action. The role-based languages we consider provide support only to one of the two steps: for instance, TPL (a role-assignment policy language) policies describe to which role the requester can be mapped; this role must then be fed as input to an existing role-based access control mechanism. A similar approach is taken by Cassandra and *RT*. On the other hand Ponder (authorization) policies are meant to support the second step, i.e., they allow to define

which actions may be performed by a requester who has already been successfully authenticated.

Role-based authentication mechanisms require that the requester provides some information in order to map her to some role(s). In the easiest case this information can be once again a (uid, pwd) pair, but systems which need a stronger authentication usually exploit credentials, i.e., digital certificates representing statements certified by given entities (*certification authorities*) which can be used in establishing properties of their holder. More modern approaches (e.g., EPAL, WSPL and XACML) directly exploit the properties of the requester in order to make an authorization decision, i.e., they do not split the authorization process in two parts like role-based languages. Nevertheless they do not use credentials in order to certify the properties of the requester.

Credentials, as well as declarations (i.e., not signed statements about properties of the holder) are however supported by PeerTrust, Protune and PSPL, which are languages designed to support the trust negotiation (Winsborough et al., 2000) vision: trust between peers is established by exchanging sets of credentials between them in a negotiation which may consist of several steps.

3.2 Policy types

Policies can be exploited in a number of fields and with different goals: security, management, conversation, quality-of-service, quality-of-protection, reliable messaging, reputation-based, provisional policies are just some examples of policies which are encountered in the literature. Here we focus on policy types which will be mentioned in the following, for instance because some language we consider has been explicitly designed to support that kind of policy.

Role-assignment policies As the name suggests, role-assignment policies specify which conditions a requester must fulfill in order to belong to some server-defined role. Role-assignment policies are typically used in role-based policy languages like Cassandra, *RT* and TPL which postulate the existence of a back-end role-based access control mechanism to which the role will be fed in order to perform the actual authorization

Access control policies Access control is concerned with limiting the activities a user is allowed to perform. Consequently access control policies define the prerequisites the requester must fulfill in order to have the activity she asked for performed

Privacy policies Privacy policies are meant to protect the privacy of the user: they need to reflect

current regulations and possibly promises made to the customers. Privacy policies arise further issues in comparison to access control policies, as they require a more sophisticated treatment of deny rules and conditions on context information; moreover privacy policy languages have to take into account the notion of “purpose”, which is essential to privacy legislation. A subset of privacy policies are *enterprise* privacy policies which furthermore have to provide support to more restrictive enterprise-internal practices and may need to handle customer preferences. EPAL was especially designed in order to target enterprise privacy policies

Obligation policies Obligation policies specify the actions that must be performed when certain events occur, i.e., they are event-triggered condition-action rules. Obligation policies may be exploited, e.g., to specify which actions must be performed when security violations occur or under which circumstances auditing and logging activities have to be carried out. Obligation policies are supported, among others, by KAoS, Ponder and Rei

4 Presentation of the considered policy languages

To date a bunch of policy languages have been developed and are currently available: we have chosen those which at present seem to be the most popular ones, namely Cassandra (Becker and Sewell, 2004), EPAL (Ashley et al., ; ?), KAoS (Uszok et al., 2003), PeerTrust (Gavriloaie et al., 2004), Ponder (Damianou et al., 2001), Protune (Bonatti et al., 2006; ?), PSPL (Bonatti and Samarati,), Rei (Kagal et al., 2003), *RT* (Li and Mitchell,), TPL (Herzberg et al., 2000), WSPL (Anderson, 2004) and XACML (Lorch et al., ; ?). The information we will provide about the aforementioned languages is based on the referenced documents. Whenever a feature we are going to tackle is not addressed in the considered literature nor is it known to the authors in other way, the feature is supposed not to be provided by the language.

The number and variety of policy languages proposed so far is justified by the different requirements they had to accomplish and the different use cases they were designed to support. Ponder was meant to help local security policy specification and security management activities, therefore typical addressed application scenarios include registration of users or logging and audit events, whereas firewalls, operating

systems and databases belong to the applications targeted by the language. WSPL's name itself (namely Web Services Policy Language) suggests its goal: supporting description and control of various aspects and features of a web service. Web services are addressed by KAoS too, as well as general-purpose grid computing, although it was originally oriented to software agent applications (where dynamic runtime policy changes need to be supported). Rei's design was primarily concerned with support to pervasive computing applications (i.e. those in which people and devices are mobile and use wireless networking technologies to discover and access services and devices). EPAL (Enterprise Privacy Authorization Language) was proposed by IBM in order to support enterprise privacy policies. Some years before IBM had already introduced the pioneer role-based policy language TPL (Trust Policy Language), which paved the way to other role-assignment policy languages like Cassandra and RT (Role-based Trust-management framework), both of which aimed to address access control and authorization problems which arise in large-scale decentralized systems when independent organizations enter into coalitions whose membership and very existence change rapidly. The main goal of PSPL (Portfolio and Service Protection Language) was providing a uniform formal framework for regulating service access and information disclosure in an open, distributed network system like the web; support to negotiations and private policies were among the basic reasons which led to its definition. PeerTrust is a simple yet powerful language for trust negotiation on the Semantic Web based on a distributed query evaluation. Trust negotiation is addressed by Protune too, which supports a broad notion of "policy" and does not require shared knowledge besides evidences and a common vocabulary. Finally XACML (eXtensible Access Control Markup Language) was meant to be a standard general purpose access control policy language, ideally suitable to the needs of most authorization systems.

Given the multiplicity of available languages and the sometimes very specific contexts they fit into, one may argue that a meaningful comparison among them is impossible or, at least, meaningless. We claim that such a comparison is not only possible but even worth: to this aim we identified ten criteria which should be taken into account in designing every policy language. By comparing the choices designers made in addressing such criteria, useful conclusions can be drawn about strong points and weaknesses of each policy language. More important yet, by outlining advantages and drawbacks of each language, our analysis will hopefully help a user in choosing the one

which mostly suits her needs.

5 Presentation of the considered criteria

We acknowledge the remark made by (Duma et al.,), according to which a comparison among policy languages on the basis of the criteria presented in (Seamons et al., 2002) is only partially satisfactory for a designer, since general features do not help in understanding which kind of policies can be practically expressed with the constructs available in a language. Therefore in our comparison we selected a good deal of criteria having a concrete relevance (e.g., whether actions can be defined within a policy and executed during its evaluation, how the result of a request looks like, whether the language provides extensibility mechanisms and to which extent . . .). On the other hand, since we did not want to come short on theoretical issues, we selected four additional criteria, basically taken from (Seamons et al., 2002) and somehow reworked and updated them. We called these more theoretical criteria *core policy properties* whereas more practical issues have been grouped under the common label *contextual properties*.

5.1 Core policy properties

Well-defined semantics According to (Seamons et al., 2002) we consider a policy language's semantics to be well-defined if the meaning of a policy written in that language is independent of the particular implementation of the language. Logic programs and Description logic knowledge bases have a mathematically defined semantics, therefore we assume policy languages based on either of the two formalisms to have well-defined semantics

Monotonicity In the sense of logic a system is monotonic if the set of conclusions which can be drawn from the current knowledge base does not decrease by adding new information to the knowledge base. In the sense of (Seamons et al., 2002) a policy language is considered to be monotonic if an accomplished request would also be accomplished if accompanied by additional disclosure of information by the peers: in other words, disclosure of additional evidences and policies should only result in the granting of additional privileges. Policy languages may be not monotonic in the sense of logic (as it happens with Logic programming-based languages) but still be mono-

tonic in the sense of (Seamons et al., 2002), like Protune

Condition expressiveness A policy language must allow to specify under which conditions the request of the user (e.g., for performing an action or for disclosing a credential) should be accomplished. Policy languages differ in the expressiveness of such conditions: some languages allow to set constraints on properties of the requester, but not on parameters of the requested action, moreover constraints on environmental factors (e.g., time) are not always supported. Cassandra's expressiveness can be even tuned by varying the constraint domain it is equipped with. This criterion subsumes "credential combinations", "constraints on attribute values" and "inter-credential constraints" in (Seamons et al., 2002)

Underlying formalism A good deal of policy languages base on some well-known formalism: Protune and PSPL base on Logic programming, whereas Cassandra, Peer-Trust and RT on a subset of it (namely Constrained DATALOG) and KAoS on Description logics. Knowledge about the formalism a language bases upon can be useful in order to understand some basic features of the language itself: e.g., the fact that a language is based on Logic programming with negation (as failure) entails consequences regarding the monotonicity of the language (in the sense of logic), whereas knowing that Description logic knowledge bases may contain contradictory statements could induce to infer that a Description logics-based language needs a way to deal with such contradictions (as it happens with KAoS and Rei)

5.2 Contextual properties

Action execution During the evaluation of a policy some actions may have to be performed: one may want to retrieve the current system time (e.g., in case authorization should be allowed only in a specific time frame), to send a query to a database or to record some information in a log file. Cassandra equipped with a suitable constraint domain allows to specify side-effect free actions, whereas e.g., Ponder and XACML allow some kind of actions. It is worth noticing that this criterion evaluates whether a language allows the *policy writer* to specify actions within a policy: during the evaluation of a policy the engine may carry out non-trivial actions on its own (e.g., both RT and TPL engines provide automatic resolution of credential chains) but such actions are not considered in our investigation

Delegation Delegation is often used in access control systems to cater for temporary transfer of access rights to agents acting on behalf of other ones (e.g., passing write rights to a printer spooler in order to print a file). The right of delegating is a right as well and as such can be delegated, too. Some languages provide a means for cascaded delegations up to a certain length, whereas others allow unbounded delegation chains. In order to support delegation many languages provide a specific built-in construct, whereas others (e.g., Cassandra and Protune) exploit more fine-grained features of the language in order to simulate high-level constructs. The latter approach allows to support more flexible delegation policies and is hence more suited for expressing the subtle but significant semantic differences which appear in real-world applications

Type of evaluation

Evidences During the evaluation of authentication policies, it may be needed for the requester to provide some signed statements (*credentials*) issued by a trusted entity and asserting some properties about the requester itself. Credentials are not supported by languages not targeting authentication policies (e.g., Ponder) nor by e.g., EPAL or KAoS. PeerTrust, Protune and PSPL provide another kind of evidence, namely *declarations* which are non-signed statements about properties of the holder (e.g., credit-card numbers)

Negotiation support (Anderson, 2004) adopts a broad notion of "negotiation", namely a negotiation is supposed to happen between two peers whenever (i) both peers are allowed to define a policy and (ii) both policies are taken into account when processing a request. According to this definition, WSPL supports negotiations as well. In this paper we adopt a narrower definition of negotiation by adding a third prerequisite stating that (iii) the evaluation of the request must be distributed, i.e., both peers must locally evaluate the request and either decide to terminate the negotiation or send a partial result to the other peer who will go on with the evaluation. Whether the evaluation is local or distributed may be considered an implementation issue, as long as policies are freely disclosable. Distributed evaluation is required under a conceptual point of view as soon as the need for keeping policies private arises: indeed if policies were not private, simply merging the peers' policies would reveal possible compatibilities between them

Policy engine decision The result of the evaluation

of a policy must be notified to the requester. The result sent back by the policy engine may carry information to different extents: in the easiest case a boolean answer may be sent (allowed vs. denied). Some languages (e.g., EPAL, WSPL and XACML) support error messages, whereas Cassandra returns a set of constraints which is a subset of the one in the requester's query. Protune is the only language providing enough informative content to let the user understand how the result was computed (and thereby why the query succeeded/failed)

Extensibility Since experience shows that each system needs to be updated and extended with new features, a good programming practice requires to keep things as general as possible in order to support future extensions. Almost every language provides some support to extensibility. RT may be regarded as an exception since, as pointed out by (Becker and Sewell, 2004), the need for more advanced features was handled by releasing a new flavor of the language (available RT flavors can be obtained by combining RT_0 and RT_1 on the one hand with RT^T and/or RT^D on the other one). In the following we will provide a description of the mechanisms languages adopt in order to support extensibility

6 Comparison

In this section the considered policy languages will be compared according to the criteria outlined in section 5. The overall results of the comparison are summarized in Table 6.

Well-defined semantics A policy language's semantics is well-defined if the meaning of a policy written in that language is independent of the particular implementation of the language. We assume policy languages based on Logic programming or Description logics to have well-defined semantics: the formalisms underlying the considered policy languages will be accounted for in the following. So far we restrict ourselves to list the languages provided with a well-defined semantics, namely, Cassandra, EPAL, KAoS, PeerTrust, Protune, PSPL, Rei and RT

Monotonicity In the sense of (Seamons et al., 2002) a policy language is considered to be monotonic if disclosure of additional evidences and policies only results in the granting of additional privileges, therefore the concept of "monotonicity" does not apply to languages which do not pro-

vide support for credentials, namely EPAL, Ponder, WSPL and XACML. All other languages are monotonic, with the exception of TPL, which explicitly chose to support *negative certificates*, stating that a user can be assigned a role R if there exists no credential of some type claiming something about it. The authors of TPL acknowledge that it is almost impossible proving that there does not exist such a credential somewhere, therefore they interpret their statement in a restrictive way, i.e., they assume that such a credential does not exist if it is not present in the local repository. Despite this restrictive definition the language is not monotonic since, as soon as such a credential is released and stored in the repository, consequences which could be previously drawn cannot be drawn anymore

Condition expressiveness A role-based policy language maps requesters to roles, the assigned role is afterwards exploited in order (not) to authorize the requester to execute some actions. The mapping to a role may in principle be performed according to the identity or other properties of the requester (to be stated by some evidence) and eventually environmental factors (e.g., current time). Cassandra (equipped with a suitable constraint domain) supports both scenarios. Environmental factors are not taken into account by TPL, where the mapping to a role is just performed according to the properties of the requester; such properties can be combined by using boolean operators, moreover a set of built-in operators (e.g., greater than, equal to) is provided in order to set constraints on their values. Environmental factors are not taken into account by RT_0 either, where role membership is identity-based, meaning that a role must explicitly list its members; nevertheless since (i) roles are allowed to express set of entities having a certain property and (ii) conjunctions and disjunctions can be applied to existing roles in order to create new ones, then role membership is finally based on properties of the requester. RT_1 goes a step beyond and, by adding the notion of *parametrized role*, allows to set constraints not only on properties of the requester but even on the ones of the object, the requested action should be performed upon; the last feature makes the second step traditional role-based policy languages consist of unnecessary, therefore RT_1 , as well as the other RT flavors basing on it, may be considered to lay on the border between role-based and non role-based policy languages. A non role-based policy language does not split the authentication process in two different steps

but directly provides an answer to the problem whether the requester should be allowed to execute some action. In this case the authorization decision can be made in principle not only depending on properties of the requester or the environment, but also according to the ones of the object the action would be performed upon as well as parameters of the action itself. EPAL introduces the further notion of “purpose” for which a request was sent and allows to set conditions on it. Some non role-based languages make a distinction between conditions which must be fulfilled in order for the request to be taken into consideration (which we call *prerequisites*, according to the terminology introduced by (Bonatti and Samarati,)) and conditions which must be fulfilled in order for the request to be satisfied (*requisites* according to (Bonatti and Samarati,)); not always both kinds of conditions have the same expressiveness. Let start checking whether and to which extent the non role-based policy languages we considered support prerequisites: WSPL and XACML allow only to use a simple set of criteria to determine a policy’s applicability to a request, whereas Ponder provides a complete solution which allows to set prerequisites involving properties of requester, object, environment and parameters of the action. Prerequisites can be set in EPAL and PSPL as well; the expressiveness of PSPL prerequisites is the same as the one of its requisites, which we will discuss later. With the exception of Ponder, which allows restrictions on the environment just for delegation policies, each other language supports requisites (Rei is even redundant in this respect): KAoS allows to set constraints on properties of the requester and the environment, Rei also on action parameters and Protune, PSPL, WSPL and XACML also on properties of the object. EPAL supports conditions on the purpose for which a request was sent but not on environmental factors. Attributes must be typed in EPAL, WSPL, XACML and typing can be considered a constraint on the values the attribute can assume, anyway the definition of the semantics of such attributes is outside WSPL’s scope. Finally, in PeerTrust conditions can be expressed by setting guards on policies: each policy consists of a guard and a body, the body is not evaluated until the guard is satisfied

Underlying formalism The most part of languages provided with a well-defined semantics rely on some kind of Logic programming or Description logics. Logic programming is the semantic foundation of Protune and PSPL, whereas a subset

of it, namely Constraint DATALOG, is the basis for Cassandra, PeerTrust and *RT*. KAoS relies on Description logics, whereas Rei combines features of Description logics (ontologies are used in order to define domain classes and properties associated with the classes), Logic programming (Rei policies are actually particular Logic programs) and Deontic logic (in order to express concepts like rights, prohibitions, obligations and dispensations). EPAL exploits Predicate logic without quantifiers. Finally, no formalisms underly Ponder (which only bases on the Object-oriented paradigm), TPL, WSPL and XACML

Action execution Ponder allows to access system properties (e.g., time) from within a policy, moreover it supports obligation policies, asserting which actions should be executed if some event happens: examples of such actions are printing a file, tracking some data in a log file and enabling/disabling user accounts. XACML allows to specify actions within a policy; these actions are collected during the policy evaluation and executed before sending a response back to the requester. A similar mechanism is provided by EPAL and of course by WSPL, which is indeed a specific profile of XACML. The only actions which the policy writer may specify in PeerTrust and PSPL are related to the sending of evidences, whereas Protune supports whatever kind of actions, not necessarily side-effect free, as long as a basic assumption holds, namely that action results do not interfere with each other (i.e., that actions are independent). Cassandra (equipped with a suitable constraint domain) allows to call side-effect free functions (e.g., to access the current time). It is worth noticing that languages allowing to specify actions within policies can to some extent simulate obligation policies, as long as the triggering event is the reception of a request, although the flexibility provided by Ponder is not met in such languages. Finally, KAoS, Rei, *RT* and TPL do not support execution of actions

Delegation Ponder defines a specific kind of policies in order to deal with delegation: the field *valid* allows *positive* delegation policies to specify constraints (e.g., time restrictions) to limit the validity of the delegated access rights. Rei allows not only to define policy delegating rights but even policy delegating the right to delegate (some other right). Delegation is supported by *RT^D* (“D” stands indeed for “delegation”): being *RT* a role-based language, the right which can be delegated is the one of activating a role, i.e., the possibility of acting as a member of such a role. Ponder delegation

chains have length 1, whereas in *RT* delegation chains always have unbounded length. Cassandra and Protune provide a more flexible mechanism which allows to explicitly set the desired length of a delegation chain (as well as other properties of the delegation): in order to obtain such a flexibility the aforementioned languages do not provide high-level constructs to deal with delegation but simulate them by exploiting more fine-grained features of the language. Delegation (of authority) can be expressed in PeerTrust by exploiting operator “@”. Finally, EPAL, KAoS, PSPL, TPL, WSPL and XACML do not support delegation

Type of evaluation The most part of the considered languages require that all policies to be evaluated are collected in some place before starting the evaluation, which is hence performed locally: this is the way EPAL, KAoS, Ponder, *RT* and TPL work. Other languages, namely Cassandra, Rei, WSPL and XACML, perform policy evaluation locally, nevertheless they provide some facility in order to collect policies (or policy fragments) which are spread over the net: e.g., in XACML combining algorithms define how to take results from multiple policies and derive a single result, whereas Cassandra allows policies to refer to policies of other entities, so that policy evaluation may trigger queries of remote policies (possibly the requester’s one) over the network. Policies can be collected into a single place if they are freely disclosable (assuming that the place they are collected into is not a trusted one), therefore the languages mentioned so far do not address the possibility that policies themselves may have to be kept private. Protection of sensitive policies can be obtained only by providing support to distributed policy evaluation, like the one carried out by PeerTrust, Protune or PSPL

Evidences The result of a policy’s evaluation may depend on the identities or other properties of the peer who requested for its evaluation: a means hence to be provided in order for the peers to communicate such properties to each other. Such information is usually sent in the form of digital certificates signed by trusted entities (*certification authorities*) and called *credentials*. Credentials are a key element in Cassandra, *RT* and TPL, whereas they are unnecessary in Ponder, whose policies are concerned with limiting the activity of users who have already been successfully authenticated. The authors of PSPL were the first ones advocating for the need of exchanging non-signed statements (e.g., credit card numbers), which they called *declarations*; declarations are

supported by PeerTrust and Protune as well. Finally, EPAL, KAoS, Rei, WSPL and XACML do not support evidences

Negotiation support As stated above, we use a narrower definition of negotiation than the one provided in (Anderson, 2004), into which WSPL does not fit, therefore only pretty few languages support negotiation in the sense we specified above, namely Cassandra, PeerTrust, Protune and PSPL

Policy engine decision The evaluation of a policy should end up with a result to be sent back to the requester. In the easiest case such result is a boolean stating whether the request was (not) accepted (and thereby accomplished): KAoS, PeerTrust, Ponder, PSPL, *RT* and TPL conform to this pattern. Besides permit and deny WSPL and XACML provide two other result values to cater for particular situations: `not_applicable` is returned whenever no applicable policies or rules could be found, whereas `indeterminate` accounts for some error which occurred during the processing; in the latter case optional information is available to explain the error. A boolean value, stating whether the request was (not) fulfilled, does not make sense in the case of an obligation policy, which simply describes the actions which must be executed as soon as an event (e.g., the reception of a request) happens, therefore besides the so-called *rulings* allow and deny EPAL defines a third value (`don’t_care`) to be returned by obligation policies; one of the elements an EPAL policy consists of is a global condition which is checked at the very beginning of the policy evaluation: not fulfilling such a condition is considered an error and a corresponding error message (`policy_error`) is returned; a further message (`scope_error`) is returned in case no applicable policies were found. Cassandra’s request format contains (among others) a set of constraints c belonging to some constraint domain; the response consists of a subset c' of c which satisfies the policy; in case $c' = c$ (resp. c' is the empty set) `true` (resp. `false`) is returned. Protune allows for more advanced explanation capabilities: not only is it possible to ask why (part of) a request was (not) fulfilled (`why` and `why-not` queries respectively), but the requester is even allowed to ask since the beginning which steps she has to perform in order for her request to be accomplished (`how-to` and `what-if` queries). A rudimentary form of `what-if` queries is supported also by Rei obligation policies: the requester can decide whether to complete the obli-

gation by comparing the effects of meeting the obligation (`MetEffects`) and the effects of not meeting the obligation (`NotMetEffects`)

Extensibility Extensibility is a fuzzy concept: almost all languages provide some extension points to let the user adapt the language to her current needs, nevertheless the extension mechanism greatly varies from language to language: here we will briefly summarize the means the various languages provide in order to address extensibility. Extensibility is described as one of the criteria taken into account in designing Ponder: in order to provide smoothly support to new types of policies that may arise in the future, inheritance was considered a suitable solution and Ponder itself was therefore implemented as an object-oriented language. XACML's support to extensibility is two-fold: (i) on the one hand new datatypes, as well as functions for dealing with them, may be defined in addition to the ones already provided by XACML. Datatypes and functions must be specified in XACML requests, which indeed consists of typed attributes associated with the requesting subjects, the resource acted upon, the action being performed and the environment (ii) as we mentioned above, XACML policies can consist of any number of distributed rules; XACML already provides a number of combining algorithms which define how to take results from multiple policies and derive a single result, nevertheless a standard extension mechanism is available to define new algorithms. Using non-standard user-defined datatypes would lead to wasting one of the strong points of WSPL, namely the standard algorithm for merging two policies, resulting in a single policy that satisfies the requirements of both (assuming that such a policy exists), since there can be no standard algorithm for merging policies exploiting user-defined attributes (except where the values of the attributes are exactly equal); use of non-standard algorithms would in turn mean that the policies could not be supported using a base standard policy engine. Being standardization the main goal of WSPL, no wonder that it comes short on the topic "extensibility", which is not necessarily a drawback, if the assertion of (Anderson, 2004) holds: "most web services will probably use fairly simple policies in their service definitions". Ontologies are the means to cater for extensibility in KAoS and Rei: the use of ontologies facilitates a dynamic adaptation of the policy framework by specifying the ontology of a given environment and linking it with the generic framework ontology; both KAoS and Rei define

basic built-in ontologies, which are supposed to be further extended for a given application. Extensibility was the main issue taken into account in the design of Cassandra: its authors realized that standard policy idioms (e.g., role hierarchy or role delegation) occur in real-world policies in many subtle variants: instead of embedding such variants in an *ad hoc* way, they decided to define a policy language able to express this variety of features smoothly; in order to achieve this goal, the key element is the notion of *constraint domain*, an independent module which is plugged into the policy evaluation engine in order to adjust the expressiveness of the language; the advantage of this approach is that the expressiveness (and hence the computational complexity) of the language can be chosen depending on the requirements of the application and can be easily changed without having to change the language semantics. A standard interface to external packages is the means provided by Protune in order to support extensibility: functionalities of a component implementing such interface can be called from within a Protune policy. Finally, PeerTrust, PSPL, *RT* and TPL do not provide extension mechanisms

7 Discussion

In this section we review the comparison performed in section 6 and provide some general comments.

By carrying out the task of comparing a considerable amount of policy languages, we came to believe that they may be classified in two big groups collecting, so to say, *standard-oriented* and *research-oriented* languages respectively. EPAL, WSPL and XACML can be considered standard-oriented languages since they provide a well-defined but restricted set of features: although it is likely that this set will be extended as long as the standardization process proceeds, so far the burden of providing advanced features is charged on the user who need them; standard-oriented languages are hence a good choice for users who do not need advanced features but for whom compatibility with standards is a foremost issue. Ponder, *RT* and TPL are somehow placed in between: on the one hand Ponder provides a complete authorization solution, which however takes place after a previously overcome authentication step, therefore Ponder cannot be applied to contexts (like pervasive environments) where users cannot be accurately identified; on the other hand *RT* and TPL do not provide a complete authorization solution, since they can only

	Cassandra	EPAL	KAoS	PeerTrust	Ponder	Protune	PSPL	Rei	RT	TPL	WSPL	XACML
Well-defined semantics	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	No
Monotonicity	Yes	–	Yes	Yes	–	Yes	Yes	Yes	Yes	No	–	–
Underlying formalism	Constraint DATALOG	Predicate logic without quantifiers	Description logics	Constraint DATALOG	Object-oriented paradigm	Logic programming	Logic programming	Deontic logic, Logic programming, Description logics	Constraint DATALOG	–	–	–
Action execution	Yes (side-effect free)	Yes	No	Yes (only sending evidences)	Yes (access to system properties)	Yes	Yes (only sending evidences)	No	No	No	Yes	Yes
Delegation	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes (RT^D)	No	No	No
Type of evaluation	Distributed policies, Local evaluation	Local	Local	Distributed	Local	Distributed	Distributed	Distributed policies, Local evaluation	Local	Local	Distributed policies, Local evaluation	Distributed policies, Local evaluation
Evidences	Credentials	No	No	Credentials, Declarations	–	Credentials, Declarations	Credentials, Declarations	–	Credentials	Credentials	No	No
Negotiation	Yes	No	No	Yes	No	Yes	Yes	No	No	Yes	No (policy matching supported)	No
Result format	A/D and a set of constraints	A/D, scope error, policy error	A/D	A/D	A/D	Explanations	A/D	A/D ^a	A/D	A/D	A/D, not applicable, indeterminate	A/D, not applicable, indeterminate
Extensibility	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	No	Yes

Table 1: Policy language comparison (“–” = not applicable)

^aFor obligation policies a rough version of “What-if” query is available.

map requesters to roles and need to rely on some external component to perform the actual authentication (although parametrized roles available in RT_1 and the other RT flavors basing on it make the previous statement no longer true). Finally research-oriented languages strive toward generality and extensibility and provide a number of more advanced features in comparison with standard-oriented languages (e.g., conflict harmonization in KAoS and Rei, negotiations in Cassandra, PeerTrust and PSPL or explanations in Protune); they should be hence the preferred choice for users who do not mind about standardization issues but require the advanced functionalities that research-oriented languages provide.

8 Conclusions

Policies are a well-known approach to protecting security and privacy of users in the context of the Semantic Web. In the last years a number of policy languages were proposed to address different application scenarios. In order to help the user in choosing the language best suiting her needs, policy language comparisons were proposed in the literature. Nevertheless available comparisons are either out-of-date or too narrow in order to provide the users with a broader picture of the research field. In this paper we considered twelve relevant policy languages and compared them on the strength of ten criteria which should be taken into account in designing every policy language. Some criteria were already known in the literature, others were introduced in our work for the first time. By comparing the choices designers made in addressing such criteria, useful conclusions can be drawn about strong points and weaknesses of each policy language. More important yet, having outlined advantages and drawbacks of each language, our analysis will hopefully help a user in choosing the one which mostly suits her needs.

REFERENCES

- Anderson, A. H. A comparison of two privacy policy languages: Epal and xacml. In *Proceedings of the 3rd ACM workshop on Secure web services*, pages 53–60. ACM Press.
- Anderson, A. H. (2004). An introduction to the web services policy language (wspl). In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 189–192. IEEE Computer Society.
- Ashley, P., Hada, S., Karjoth, G., Powers, C., and Schunter, M. Enterprise privacy authorization language (epal 1.2).
- Becker, M. Y. and Sewell, P. (2004). Cassandra: Distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, pages 159–168, Yorktown Heights, NY, USA. IEEE Computer Society.
- Bonatti, P., Olmedilla, D., and Peer, J. (2006). Advanced policy explanations. In *17th European Conference on Artificial Intelligence (ECAI 2006)*, Riva del Garda, Italy. IOS Press.
- Bonatti, P. and Samarati, P. Regulating service access and information release on the web. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 134–143. ACM Press.
- Damianou, N., Dulay, N., Lupu, E., and Sloman, M. (2001). The ponder policy specification language. In *2nd IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 18–38. Springer.
- Duma, C., Herzog, A., and Shahmehri, N. Privacy in the semantic web: What policy languages have to offer. In *Eighth IEEE International Workshop on Policies for Distributed Systems and Networks-TOC (POLICY)*, pages 5–8. IEEE Computer Society.
- Gavriloaie, R., Nejdl, W., Olmedilla, D., Seamons, K. E., and Winslett, M. (2004). No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *1st European Semantic Web Symposium (ESWS 2004)*, volume 3053 of *Lecture Notes in Computer Science*, pages 342–356, Heraklion, Crete, Greece. Springer.
- Herzberg, A., Mass, Y., Michaeli, J., Ravid, Y., and Naor, D. (2000). Access control meets public key infrastructure, or: Assigning roles to strangers. In *2000 IEEE Symposium on Security and Privacy*, pages 2–14. IEEE Computer Society.
- Kagal, L., Finin, T. W., and Joshi, A. (2003). A policy language for a pervasive computing environment. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 63–, Lake Como, Italy. IEEE Computer Society.
- Li, N. and Mitchell, J. C. Rt: A role-based trust-management framework. In *Third DARPA Information Survivability Conference and Exposition (DIS-CEX III)*.
- Lorch, M., Proctor, S., Lepro, R., Kafura, D., and Shah, S. First experiences using xacml for access control in distributed systems. In *Proceedings of the 2003 ACM workshop on XML security*, pages 25–37. ACM Press.
- Seamons, K. E., Winslett, M., Yu, T., Smith, B., Child, E., Jacobson, J., Mills, H., and Yu, L. (2002). Requirements for policy languages for trust negotiation. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 68–79, Monterey, CA, USA. IEEE Computer Society.
- Tonti, G., Bradshaw, J. M., Jeffers, R., Montanari, R., Suri, N., and Uszok, A. (2003). Semantic web languages

for policy representation and reasoning: A comparison of kaos, rei, and ponder. In *2nd International Semantic Web Conference (ISWC)*, volume 2870 of *Lecture Notes in Computer Science*, pages 419–437, Sanibel Island, FL, USA. Springer.

- Uzbek, A., Bradshaw, J. M., Jeffers, R., Suri, N., Hayes, P. J., Breedy, M. R., Bunch, L., Johnson, M., Kulkarni, S., and Lott, J. (2003). Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 93–96, Lake Como, Italy. IEEE Computer Society.
- Winsborough, W., Seamons, K., and Jones, V. (2000). Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, pages 88–102. IEEE Computer Society.