A Flexible Policy-Driven Negotiation Model

Juri L. De Coi, Daniel Olmedilla
L3S Research Center & Hannover University
Hannover, Germany
{decoi, olmedilla}@L3S.de

Abstract

*Policy-driven negotiations are gaining interest among the research community. A large number of policy languages with different expressiveness have been developed in order to suit different scenarios. This paper summarizes the general requirements a negotiation framework must cover and presents a flexible negotiation model that addresses all these requirements and subsumes existing models to date. An instantiation of this model and an architecture with reusable components that integrates two existing trust negotiation languages (*PEERTRUST *and* PROTUNE*) are provided.*

1 Introduction

During the last decade, the amount of users with Internet access has dramatically grown all over the world. This situation has boosted the Web but also provoked the boom of other kinds of distributed environments like Peer-to-Peer networks and Grid environments. Among all of them, new security requirements arose due to the high amount and heterogeneity of potential users. While previously the set of potential users accessing a system was mainly restricted to those already known, now any two strangers should be able to communicate and perform transactions with such systems. Traditional authorization relies on the fact that client identities can be mapped to a set of permissions. This authorization process is not applicable anymore since clients may not be known in advance.

A new authorization scheme called Trust Negotiation has emerged and is gaining interest among the research community. It allows two strangers to bilaterally establish trust in an incremental process. This process is driven by statements that specify what is released and under which conditions. These statements are generally called policies and many languages have been developed to date [2, 6, 5, 4, 8, 12] in order to represent them. These languages differ on semantics and expressiveness since they were developed in order to suit different scenarios. Additionally, their semantics normally implicitly hints a negotiation model, which varies from one language to another.

In this paper, we describe the requirements a general negotiation model must address, including some that are not considered in previous trust negotiation languages. In addition, we present and describe in detail a general negotiation model which addresses those requirements and subsumes existing models to date.

This paper is organized as follows: after a brief introduction into the motivation and description of trust negotiation in section 2, section 3 highlights the main concepts that should be held in mind when designing a trust negotiation framework. A review of how these concepts are addressed by current state of the art on negotiation models is provided in section 4. Section 5 describes a flexible negotiation model, which addresses the identified requirements and subsumes other existing models. Section 6 describes the negotiation algorithm which instantiates the model presented as well as the system architecture in which it is implemented. Finally section 7 presents some conclusions and outlines some future work.

2 Trust Negotiation

Open distributed environments like the World Wide Web, P2P networks or Grids offer easy sharing of information, and existing protection of sensitive resources is typically based on the assumption that a requester is already known to the server (e.g., by means of previous registration and user/password authentication mechanisms). This way, the server is able to map the requester into a permission table in order to grant or deny access to a resource.

Nowadays, simple identity-based authorization mechanisms are no longer desirable in open distributed environments. Specifically, the Web provides an environment where parties may interact without being previously known to each other. In many cases, before any meaningful interaction starts, a certain level of trust must be established from scratch. Generally, trust is established through exchange of

information between the two parties, since both parties may have sensitive information that they are reluctant to disclose until the other party has proved to be trustworthy at a certain level.

To make controlled sharing of resources easy in such an environment, parties will need software that automates the process of iteratively establishing bilateral trust based on the parties' access control policies, i.e., *trust negotiation* [10] software. This software should allow every party to define policies to control outsiders' access to their sensitive resources. Policies describe what properties a party must possess (e.g., ownership of a driver license or holding a citizen id of the European Union) in order to gain access to a resource.

3 Negotiation Requirements

In this section we outline the main requirements a negotiation framework should take into account: some of them were already described in previous literature [5, 8, 11, 6, 2, 12, 7], others are clearly stated here for the first time.

Negotiation The ground requirement a client-server transaction should fulfill is obviously the possibility of having not just one-shot interactions between actors but bilateral negotiations which may be driven by policies in a (semi-)automatic fashion.

Actors Each negotiation implies two actors (e.g., Alice and some on-line bookshop). Interleaved negotiations should be allowed as well: this is the case when, in order to proceed, a negotiation requires that the result of another negotiation is made available. For instance, in order to advance the negotiation, an on-line bookshop may want to check whether the credit card number provided by a customer is a valid one and has credit. This implies starting a second negotiation between the bookshop and the VISA server. Only when this negotiation is finished and its result is available the first negotiation can continue.

External actions During a negotiation each actor may ask the other one for carrying out some actions (e.g., delivering a book or registering at a web site).

Notifications Each actor needs to be notified about whether the other actor performed the actions it was asked for. A notification may either need to be proved, so that the other actor can verify it (e.g., through a signed statement from the bank stating that a money transfer has been made), or not (e.g., provision of a delivery address).

Local actions During a negotiation each actor may need to carry out some actions which are not explicitly requested by the other actor. For example, it may be needed to access legacy systems (e.g., a database of users and passwords) or external entities (e.g., the VISA server) or make system calls (e.g., log a message into a file).

Action Selection Function In order to satisfy the negotiation's overall goal, an actor may be requested for following (at least) one out of n paths (e.g., in order to finalize a purchase, an on-line bookshop may request Alice either to subscribe a new account or to provide a credit card number). It may be the case that the actor does not want to perform all actions, but only a subset of them. In order to make this selection automatically, an Action Selection Function is required. Action Selection Functions can be exploited to support among others eager or parsimonious negotiations strategies [10].

Policy In general, an action (e.g., the disclosure of a credential) is performed only under certain circumstances. Therefore a means is required to specify under which conditions an action can be executed (e.g., "credit card number can be provided only to on-line shops belonging to the Better Business Bureau"). Typically such a means is a policy language.

Policy filtering Actors need to tell each other the conditions under which a requested action can be performed. However typically a whole policy does not need to be released, since some information contained in it may be irrelevant for the other actor (e.g., local actions to be performed) or sensitive (e.g., giving away that the bookshop delegates a decision to a third company may be private). In any case, the information contained in a filtered policy can be seen as a subset of the actor's local policy.

Termination Algorithm A Termination Algorithm can be seen as a means to ensure that a negotiation eventually terminates (and guarantee that it does not get looped). That a negotiation may never terminate can be shown by a simple example: imagine two actors who decide not to perform any external actions before the other one performs some. Termination Algorithms can be exploited to support more advanced negotiation strategies like e.g., "terminate the negotiation if in the last n negotiation steps the other actor did not perform any (external) actions and the last filtered policy it sent does not differ from the previous one" [3].

Explanation An actor should be able to ask and get (high level) information about the ongoing negotiation (e.g. how to buy a book at the bookshop) or finished ones (e.g., why the negotiation failed).

4 Related work

Concepts like *negotiation*, *policy* and *actor* are grass-roots ones in the field of Trust Negotiation, and hence are mentioned in most of the literature about this topic (cf. [5, 8, 11, 6, 2, 12, 7]), although not all (cf. [5, 7, 2]) take delegation into account.

In all these approaches assertion exchange (real credentials are not always implemented) is the only addressed external action, hence a need for *notifications* does not arise (credential reception is a particular notification on its own). Moreover local actions, when foreseen, usually limit to checking whether a credential is still valid.

Similarly a distinction between *Action Selection Function* and *Termination Algorithm* is usually not made, being both concepts grouped under the common label of *Negotiation Strategy* [13]. This often results in loss of generality: e.g., in [7] only the service provider can decide if and when to terminate a negotiation, while in [13] an empty message is considered to (unsuccessfully) terminate the negotiation. Moreover as shown in [3], a Termination Algorithm on its own has some distinguishing properties for the resulting negotiation, which once again suggests to consider a Termination Algorithm as an autonomous component of a negotiation framework.

All papers on trust negotiation assume that policies may be sensible (otherwise negotiations would not be needed) but they differ in the protection mechanisms they offer. [5, 8, 6, 2, 12, 7] suggest to set policies to protect policies as a whole, but fine-grained filtering of a policy, allowing to select just some parts of it to be disclosed to the other actor, is not foreseen.

[7] suggests that even a resource/service request could be a sensible resource, so that a negotiation for the disclosure of the request could be performed before the negotiation for obtaining the resource/service. In our model we simply consider the first negotiation as a normal negotiation for establishing a communication.

5 Negotiation model

This section describes a general negotiation model which addresses all the requirements presented in section 3.

Let A_1 and A_2 be the two actors involved in the negotiation (see figure 1). In the following we will assume that A_1 is the initial requester (e.g., Alice) whereas A_2 is the provider (e.g., the bookshop), i.e. A_1 is assumed to send a request to A_2 thus starting the negotiation. Notice that during the negotiation both actors A_1 and A_2 may both request or provide information to each other.

As shown above, a means (a language) is required to specify under which conditions the request of the other peer

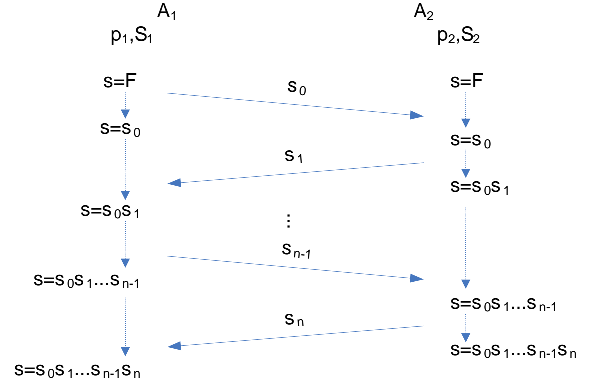

Figure 1. Sequence Diagram of a Negotiation

can be fulfilled. Let our policy language be based on normal logic program rules of the form

$$A \leftarrow L_1, \dots L_n.$$

where A is a standard logical atom (called the *head* of the rule) and $L_1, \dots L_n$ (the *body* of the rule) are literals, i.e. L_i equals either B_i or $\neg B_i$, for some logical atom B_i.

Definition 1 (Policy) *A Policy is a set of rules, such that negation is not applied to any predicate occurring in a rule's head nor to atoms representing the execution of external actions.*

This restriction ensures that policies are *monotonic* in the sense of [9], i.e. as more external actions are executed, the set of permissions does not decrease. Moreover, the restriction on negation makes policies *stratified programs*; therefore negation as failure has a clear, PTIME computable semantics that can be equivalently formulated as the perfect model semantics, the well-founded semantics or the stable model semantics [1].

Definition 2 (Negotiation Message) *A Negotiation Message is an ordered pair (p, N) where*

- $p \equiv$ *a Policy*
- $N \equiv$ *a set of notifications*

We will denote with M the set of all possible Negotiation Messages.

Definition 3 (Negotiation History) *Let A_1 and A_2 be the actors involved in a negotiation. Let A_1 be the initial requester, i.e. the sender of the first message in the negotiation. A Negotiation History σ for the actor A_j ($j = 1, 2$) is a list of Negotiation Messages*

$$\sigma_1, \dots \sigma_n \mid \sigma_i \in M$$

We will denote with $|\sigma|$ the length of σ and with σ_i the i-th element of σ. Moreover let

- $M_{snt}(\sigma) = \{\sigma_i \mid i = 2k-(j\ mod 2), 1 \leq k \leq \lfloor n/2 \rfloor\}$
- $M_{rcv}(\sigma) = \{\sigma_i \mid i = 2k-1+(j\ mod 2), 1 \leq k \leq \lfloor n/2 \rfloor\}$

denote the sequence of messages sent and received respectively.

A Negotiation History may also be referred to as Negotiation State.

Intuitively, a message sent by an actor provokes the same message to be received by the other actor. In addition, messages between actors are sent alternately, i.e. a message sent by one actor is followed by another message sent by the other actor. Notice that according to this definition, the Negotiation History σ is shared by the two actors A_1 and A_2, but the sets $M_{snt}(\sigma)$ and $M_{rcv}(\sigma)$ are swapped between them. Therefore it holds that

$$M^{A_1}_{snt}(\sigma) = M^{A_2}_{rcv}(\sigma)$$

and

$$M^{A_1}_{rcv}(\sigma) = M^{A_2}_{snt}(\sigma)$$

In order to ease the notation in the following, given a Negotiation History σ, we define the following entities

- $lp_{snt}(\sigma) = p_{i_{max}} \mid i_{max} = max\{i \mid (p_i, n_i) \in M_{snt}(\sigma)\}$
- $lp_{rcv}(\sigma) = p_{i_{max}} \mid i_{max} = max\{i \mid (p_i, n_i) \in M_{rcv}(\sigma)\}$

Intuitively, lp_{snt} (resp. lp_{rcv}) represents the last Policy sent (resp. received).

Definition 4 (Negotiation State Machine) *A Negotiation State Machine is a tuple (S, s_0, Σ, t) such that*

- $S \equiv$ *a set of Negotiation States*
- $s_0 \equiv$ *the empty list (initial state)*
- $\Sigma \equiv$ *a set of Negotiation Messages.*
- $t \equiv$ *a function $S \times \Sigma \rightarrow S$ such that given $s = (\sigma_1, \ldots \sigma_n)$ then $t(s, \sigma) = (\sigma_1, \ldots \sigma_n, \sigma)$ (transition function)*

Intuitively a Negotiation State Machine models how an actor evolves during the negotiation by exchanging messages. Σ contains both sent and received Negotiation Messages and can therefore be partitioned into two subsets Σ_{snd} and Σ_{rcv}.

Definition 5 (Negotiation Model) *A Negotiation Model is a tuple*

$$(A, P, p_0, NSM, ff, ns)$$

where

- $A \equiv$ *a set of actions*
- $P \equiv$ *a set of Filtered Policies*
- $p_0 \equiv$ *a Policy*
- $NSM \equiv$ *a Negotiation State Machine (Σ, S, s_0, t)*
- $ff \equiv$ *a function $S \rightarrow P$ (Filtering Function)*
- $ns \equiv$ *an ordered pair (asf, ta) where*
  - $asf \equiv$ *a function $S \rightarrow \mathcal{P}(A)$*
  - $ta \equiv$ *a function $S \rightarrow \{true, false\}$*

  (Negotiation Strategy)

Each occurrence of S is supposed to refer to the same set of Negotiation States.

The intended meaning is as follows

- A represents the set of external actions the actor can be asked to perform
- P represents the set of possible filtered policies
- p_0 represents the actor's local policy
- S represents the set of states in which the actor can be
- s_0 represents the initial state, i.e. the state in which the actor is at the beginning of the negotiation
- ff represents the actor's filtering algorithm
- asf represents the actor's Action Selection Function
- ta represents the actor's Termination Algorithm

Definition 6 (Bilateral Negotiation) *Let NM_1 and NM_2 be the negotiation models of A_1 and A_2 respectively. The two models NM_1 and NM_2 represent a valid bilateral negotiation if*

- *for each message sent by A_i, an identical message (i.e. with the same parameters) is received by $A_{i\ mod 2+1}$.*
- *it is allowed that a message repeats information which has previously been disclosed. However, a message containing no new information is considered empty.*
- *a negotiation model (its termination algorithm) must not allow never-ending exchange of empty messages.*

- *the information provided in the filtered policy sent by an actor in subsequent messages must be monotonic, that is, let fp_{i+2} be the result of the filtering process at step $i + 2$*
$$fp_{i+2} = ff(s_{i+2})$$
and fp_i the filtered policy sent at step i, then the following condition must hold
$$\mathcal{H}(fp_i) \subseteq \mathcal{H}(fp_{i+2})$$

6 Implementation

In this section we describe the negotiation algorithm which instantiates the model presented in section 5 as well as the architecture we have implemented in order to allow for the integration of different trust negotiation languages (we have already integrated two of them: PROTUNE and PEERTRUST) and the reuse of components among them.

6.1 Negotiation Algorithm

A general negotiation algorithm, in pseudocode form, that instantiates the model presented in previous sections is described in Figure 2.

At each negotiation step an actor (let say A_1) sends the other one (A_2) a(potentially empty) filtered policy rfp and a(potentially empty) set of notifications rn, stating the conditions to be fulfilled by A_2 as well as notifying the execution by A_1 of some actions it was asked for. As soon as A_2 receives them, it adds them to the negotiation state together with some surrounding information like the current negotiation step number and a timestamp. To filtered policies and notifications of external actions, information about whether they were sent or received is added as well, whereas notifications of local actions are equipped with information asserting that the action performed was a local one. All this further information is meant to support built-in predicates providing advanced query capabilities, like the predicate *request(RequestNumber, RequestID)* described in [4].

As soon as the negotiation state is updated, the local policy is inspected in order to identify the local actions that can be executed taking into account the new information provided in the received notifications. Those local actions are performed and as a consequence other local actions may become ready for execution. This is the case if the instantiation of a variable is a prerequisite for the execution of an action and the instantiation of this variable is (part of) the result of another action's execution like in the following policy, where the execution of $action1$ triggers $action2$ being made ready to execution.

$$goal \leftarrow action1(X), action2(X).$$

- $rfp \equiv$ Received filtered policy
- $s \equiv$ Negotiation state
- $rn \equiv$ Received notifications
- $lp \equiv$ Local policy
- $g \equiv$ Overall goal
- $oa \equiv$ Other actor
- $ta \equiv$ Termination Algorithm
- $asf \equiv$ Action Selection Function

```
add(rfp, s)
add(rn, s)
Action[] la = extractLocalActions(g, lp, s)
while(la.length != 0){
    Notification[] ln = perform(la)
    add(ln, s)
    la = extractLocalActions(g, lp, s)
}
if(isUnlocked(g, lp, s)){
    send(SUCCESS, oa)
    return
}
if(terminate(s, ta)){
    send(FAILURE, oa)
    return
}
Action[] ea = extractExternalActions(g, lp, s)
Action[] ua
for each action in ea
    if(isUnlocked(action, lp, s)) add(action, ua)
Action[] aa = selectActions(asf, ua, s)
Notification[] sn = perform(aa)
FilteredPolicy sfp = filter(g, lp, s)
add(sfp, s)
add(sn, s)
send(sfp, oa)
send(sn, oa)
```

Figure 2. Negotiation algorithm pseudocode

$$action1(_).execution : immediate.$$
$$action2(X).execution : immediate \leftarrow ground(X).$$

For this reason local action selection and execution are performed in a loop, until no more actions are ready to be executed.

As soon as a local action is performed, a notification of its execution is added to the state.

After having executed all possible local actions, A_2 now needs to check whether the whole negotiation can be already considered successful. For that purpose, the policy is evaluated in order to see whether the overall goal of the negotiation is fulfilled. If this is the case, a message is sent to A_1 telling that the negotiation can be successfully terminated. Otherwise the Termination Algorithm is consulted in order to decide whether the negotiation should continue or be terminated. According to the answer, either the negotiation goes on or a message is sent to A_1 telling that the negotiation was unsuccessfully terminated.

If the negotiation is not terminated yet, then two processes have to be performed

- A_2 filters its local policy and thereby generates the the new version of the filtered policy to be disclosed to A_1 in order to inform it about what it has to perform to advance the negotiation

- A_2 has to decide which actions requested by A_1 it will perform. Therefore, it inspects its own policy and the filtered policy received from A_1 in order to retrieve the actions requested. Since an action can be executed only if the policy protecting it is fulfilled, a check needs to be performed for each retrieved action. Those whose policies are fulfilled (*unlocked actions*) are collected and the other ones discarded.

  Unlocked actions represent potential candidates to the execution, that is, those actions which may be performed according to A_2's policy and the current negotiation state. However, just a subset of them will be actually performed, namely the one selected by the Action Selection Function

Finally, the filtered policy as well as the notifications of the performed external actions are added to the state and packed into a Negotiation Message which is sent to A_1.

6.2 Policy Framework Architecture

Since the negotiation model presented in this paper subsumes existing ones to date, it allows to support different policy languages and to integrate existing policy engines within a common architecture. We have implemented a general architecture which conforms to our negotiation model. It allows not only for co-existence of different policy engines but also for reuse of components among them such as communication interfaces or action executors. Figure 3 depicts the high level architecture of our policy Agent in which we have already integrated two different policy engines: a PROTUNE engine and a PEERTRUST engine. The Agent consists of the following modules

Network Interface Is in charge of the communication with other parties. Some example of possible interfaces are secure socket connections or web services

Internal Java API Is meant to be used by Java programs in order to integrate functionalities of the policy framework. Our implementation is distributed as a jar file, being easily included in other Java applications

Policy Engine Distributor In case more than one policy engine exists, this component is in charge of forwarding any request to the appropriate one. This way, more than one policy engine may co-exist, therefore allowing for a single agent negotiating using different policy languages (and engines)

Policy Engine A specific policy engine in charge of processing requests. Currently a PROTUNE engine and a PEERTRUST engine are implemented

Action Selection and Termination Algorithm A pluggable component specifying the general negotiation strategy of the agent (see section 3)

Inference Engine Is in charge of performing policy filtering and other evaluation processes

Execution Handler Is in charge of performing the actions specified in the policies. It contains a registry of action executors in order to map actions to the modules responsible for their execution

Credential Repository Is in charge of providing local credentials when required and checking that the credentials received during a negotiation are not forged

RDBMS Is in charge of performing queries to a relational database. This is a crucial module in order to integrate legacy systems with the policy engines

File System Is in charge of performing queries based on regular expressions on specific files in a file system

7 Conclusions and future work

Many languages with different expressiveness have been developed to date in order to provide systems with trust negotiation capabilities. However, they typically assume different features depending on the scenarios being targeted.

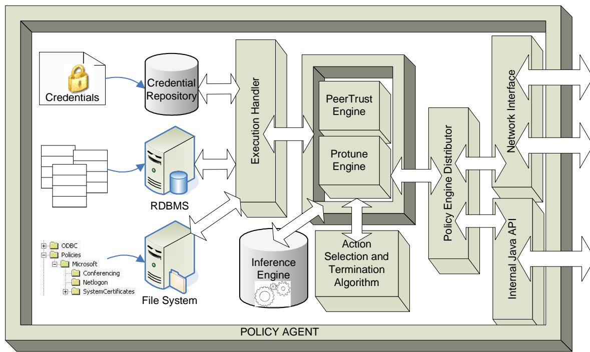

Figure 3. Policy Framework Architecture

"Policies are public and disclosed if needed", "policies are protected by other policies" or "part of the policies themselves may be protected and the rest disclosed" are just some examples. The negotiation model is greatly a consequence of the features of a policy language and therefore it is not usually described in detail. This paper reviewed existing approaches and frameworks for trust negotiation and summarizes the main features which must be covered by a general negotiation framework. In addition, a general and flexible negotiation model is presented which addresses all these features and therefore subsumes previous trust negotiation models. Finally, an instantiation of this model is presented as well as the architecture of the system implementing it.

The negotiation model (and the implemented algorithm) presented in this paper allow for arbitrary negotiation strategies to be plugged into it. We plan to explore the use of different termination algorithms and especially different action selection functions in order to (semi-)automatically perform negotiations according to user-provided sensitivity levels.

Acknowledgments

This work was partially funded by the European Commission and by the Swiss State Secretariat for Education and Research within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`). The authors would also like to thank Piero A. Bonatti for useful discussions about the topics addressed in this paper.

References

[1] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, Cambridge, 2003.

[2] M. Y. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, pages 159–168, Yorktown Heights, NY, USA, June 2004. IEEE Computer Society.

[3] P. Bonatti, T. Eiter, and M. Faella. Automated negotiation mechanisms. Technical report, Working Group I2, EU NoE REWERSE, Apr. 2006.

[4] P. A. Bonatti and D. Olmedilla. Driving and monitoring provisional trust negotiation with metapolicies. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pages 14–23, Stockholm, Sweden, June 2005. IEEE Computer Society.

[5] P. A. Bonatti and P. Samarati. Regulating service access and information release on the web. In *ACM Conference on Computer and Communications Security*, pages 134–143, 2000.

[6] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett. 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, May 2004. Springer.

[7] A. J. Lee, M. Winslett, J. Basney, and V. Welch. Traust: a trust negotiation-based authorization service for open systems. In *11th ACM Symposium on Access Control Models and Technologies*, pages 39–48, Lake Tahoe, California, USA, June 2006. ACM.

[8] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[9] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, pages 68–79, Monterey, CA, USA, June 2002. IEEE Computer Society.

[10] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated trust negotiation. DARPA Information Survivability Conference and Exposition, IEEE Press, Jan 2000.

[11] M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu. Negotiating trust on the web. *IEEE Internet Computing*, 6(6):30–37, 2002.

[12] M. Winslett, C. C. Zhang, and P. A. Bonatti. Peeraccess: a logic for distributed authorization. In *12th ACM Conference on Computer and Communications Security, CCS 2005*, pages 168–179, Alexandria, VA, USA, Nov. 2005. ACM.

[13] T. Yu, M. Winslett, and K. E. Seamons. Interoperable strategies in automated trust negotiation. In *ACM Conference on Computer and Communications Security*, pages 146–155, 2001.