EINDHOVEN UNIVERSITY OF TECHNOLOGY

An introduction to the role based trust management framework RT

Citation for published version (APA):
Czenko, M., Etalle, S., Li, D., & Winsborough, W. H. (2007). An introduction to the role based trust management framework RT. In A. Aldini, & R. Gorrieri (Eds.), *Foundations of Security Analysis and Design IV (FOSAD 2006/2007 Tutorial Lectures)* (pp. 246-281). (Lecture Notes in Computer Science; Vol. 4677). Springer. https://doi.org/10.1007/978-3-540-74810-6_9

DOI:
10.1007/978-3-540-74810-6_9

Document status and date:
Published: 01/01/2007

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 22. Aug. 2022

An Introduction to the Role Based Trust Management Framework RT

Marcin Czenko[1], Sandro Etalle[1,2], Dongyi Li[3], and William H. Winsborough[3]

[1] Department of Computer Science
University of Twente, The Netherlands
[2] University of Trento, Italy
{marcin.czenko,sandro.etalle}@utwente.nl
[3] Department of Computer Science
University of Texas
San Antonio, USA
wwinsborough@acm.org,
dli@cs.utsa.edu

Abstract. Trust Management (TM) is a novel flexible approach to access control in distributed systems, where the access control decisions are based on the policy statements, called credentials, made by different principals and stored in a distributed manner. In this chapter we present an introduction to TM focusing on the role-based trust-management framework RT. In particular, we focus on RT_0, the simplest representative of the RT family, and we describe in detail its syntax and semantics. We also present the solutions to the problem of credential discovery in distributed environments.

1 Introduction

The problem of guaranteeing that confidential data is not disclosed to unauthorized users is paramount in our IT-dominated world, and is usually tackled by implementing access control techniques. Traditional access control schemes make authorization decisions based on the identity, or the role of the requester. However, in decentralized environments, the resource owner and the requester often are unknown to one another, making access control based on identity ineffective. To give a simple example, consider the situation in which a bookstore adopts the policy of giving 10% discount to students of accredited universities. Although a certificate authority may assert that the requester's name is John Q. Smith, if this name is unknown to the bookstore, the name itself does not aid in making a decision whether he is entitled to a discount or not. What is needed is information about the rights, qualifications, and other characteristics assigned to John Q. Smith by one or more authorities (in our example, the university he attends), as well as trust information about the authority itself (e.g. is it accredited?).

Trust management [10,8,29,14,16,19,24,31] is an approach to access control in decentralized distributed systems with access control decisions based on policy statements made by multiple principals. In trust management systems, statements that are maintained in a distributed manner are often digitally signed to ensure their authenticity and integrity; such statements are called credentials or certificates. A key aspect of trust

A. Aldini and R. Gorrieri (Eds.): FOSAD 2006/2007, LNCS 4677, pp. 246–281, 2007.
© Springer-Verlag Berlin Heidelberg 2007

management is delegation: a principal may transfer limited authority over one or more resources to other principals.

RT [24,25,23] is a family of Role-based Trust management languages introduced by Li, Winsborough and Mitchell. At its most abstract, the notion of role used is simply a set of principals. The primary application of RT is intended to be authorization and access control, and the main purpose of roles is to confer to their members access to specific resources. Neverthless, roles can also be used in more general terms. For instance, membership in the role of student at the University of Texas may entail certain privileges, but serves to characterize the status of its members more generally. Such characterizations greatly facilitate granting new privileges to entire classes of users.

This tutorial is meant as an introduction to the RT family of trust management languages. It contains a thorough description of RT_0 which is the core language of the family, and some examples of the more complex members: RT_1, RT_2, RT^{T}, RT^{D} [24], and the later $\mathrm{RT}_{\ominus}$, introduced by Czenko et al. [26]. Concerning RT_0, this chapter describes in detail, syntax, semantics, decentralized storage and credential chain discovery. Technically, the content of this chapter derives directly from the original papers [24,25], with some changes which simplify the exposition while maintaining generality: in particular a restriction, we employ a restriction on queries that simplifies the definition of credential graph (see Remark 1). We also introduce new pseudo-code versions of the credential chain discovery algorithms.

2 RT_0

The RT framework encompasses a number of languages which have the same basic structure, while offering different features. The main members of the RT family are RT_0, RT_1, RT^{T}, and RT^{D}. Here we focus on the core member of RT: RT_0. Later, in Sect. 5, we are going to see examples of the features of RT_1, RT^{T}, and RT^{D}.

2.1 Syntax

The basic constructs of RT_0 are *entities*, *role names* and *roles*. *Entities* are also often called principals. They can define roles, issue credentials, and make requests. In general, an entity may be identified by a public key, or by a user account; following Li et al. [25], we abstract away from the mechanism used for identifying entities. We denote them by names starting with an uppercase letter (possibly with a subscript), e.g. A, B, B_1, and *Alice* are all entities. *Role names*, on the other hand, are denoted by strings starting with a lowercase letter (possibly with a subscript), like r, r_1, and *student*.

Finally, *roles* have the form of an entity followed by a role name, separated by a dot. For example, $A.r$, $B.r_1$, and *University.student* are valid roles. The notion of a role is central to RT_0. A role $A.r$ denotes the set of entities that are members of it – a set that we refer to informally by *members*$(A.r)$. A is called the *owner* of the role $A.r$, and is the only authority that can directly determine which are the members of $A.r$.

Permissions in RT_0 are represented by roles. For example, the permission to read confidential document on a corporate network of a company C can be represented by role *C.readConfidential*: in this case, an entity has the read permission if and only if it

belongs to *members*$(C.\mathit{readConfidential})$. Other roles are used to represent other properties, sometimes called *attributes*, that characterize the members or their relationship to the role owner. For example, membership in $C.\mathit{employee}$ might indicate an employment relationship with C. This example illustrates one aspect of how RT supports decentralization by making the entity with which one has an employment relationship explicit. In RT there is no notion of simply being employed without mentioning the entity whose judgement is being asserted or whose consent makes it so.

There are four types of credentials in RT_0 that an entity A can issue, each corresponding to a different way of defining the membership of one of A's roles, $A.r$.

- *Simple Member*: $A.r \longleftarrow D$.
  With this credential A asserts that D is a member of $A.r$.
- *Simple Inclusion*: $A.r \longleftarrow B.r_1$.
  With this credential A asserts that $A.r$ includes (all members of) $B.r_1$. This represents a delegation from A to B, as B may cause new entities to become members of the role $A.r$ by issuing credentials defining (and extending) $B.r_1$.
- *Linking Inclusion*: $A.r \longleftarrow A.r_1.r_2$.
  $A.r_1.r_2$ is called a *linked role*. With this credential A asserts that $A.r$ includes $B.r_2$ for every B that is a member of $A.r_1$. This represents a delegation from A to all the members of the role $A.r_1$.
- *Intersection Inclusion*: $A.r \longleftarrow B_1.r_1 \cap B_2.r_2$.
  $B_1.r_1 \cap B_2.r_2$ is called an *intersection*. With this credential A asserts that $A.r$ includes every principal who is a member of both $B_1.r_1$ and $B_2.r_2$. This represents partial delegations from A to B_1 and to B_2.

In the original paper introducing RT_0 [25], the number of intersection elements in the intersection inclusion credentials is unlimited. Also, each intersection element can be either a role or a linked role. Here we restrict the number of intersection elements to two and require that each intersection element be a role. This makes the description easier to follow and simplifies some definitions. However it imposes no restriction on the expressive power of the language. A credential of the more general form can be replaced by several of the more restricted credentials presented above by introducing auxiliary roles, spliting longer intersections into several intersection inclusions, and introducing a linking inclusion for each linked role.

A *policy* is a finite set of credentials. We use the term *role expression* for any entity, role, linked role, or intersection; thus each RT_0 credential has the form $A.r \longleftarrow e$, where e is a role expression. Such a credential means that *members*$(e) \subseteq$ *members*$(A.r)$. We say that this credential *defines* the role $A.r$. Further, we call A the *issuer*, e the *body* and each entity occurring syntactically in e a *subject* of this credential. To be precise, the set *base*(e) of subjects of $A.r \longleftarrow e$ is defined as follows: *base*$(A) = \{A\}$, *base*$(A.r) = \{A\}$, *base*$(A.r_1.r_2) = \{A\}$, and $base(B_1.r_1 \cap B_2.r_2) =$ *base*$(B_1.r_1) \cup$ *base*$(B_2.r_2) = \{B_1, B_2\}$.

2.2 Semantics

In this section, we present declarative semantics of RT_0. We follow Li et al. [24] and do this in terms of the semantics for logic programs by providing a translation of a policy $\mathcal{C}$

to a Datalog program, which we call the *semantic program*. The set-theoretic semantics for RT_0 can be found in [25].

Given a set $\mathcal{C}$ of RT_0 credentials (i.e. a policy) the corresponding *semantic program*, $SP(\mathcal{C})$, is a Datalog program with one ternary predicate m. Intuitively, $m(A, r, D)$ indicates that D is a member of the role $A.r$. Given an RT statement c, the *semantic program* of c, $SP(c)$, is defined as follows (identifiers starting with "?" are logical variables):

$$\begin{aligned} SP(A.r \longleftarrow D) &= m(A, r, D). \\ SP(A.r \longleftarrow B.r_1) &= m(A, r, ?X) :- m(B, r_1, ?X). \\ SP(A.r \longleftarrow A.r_1.r_2) &= m(A, r, ?X) :- m(A, r_1, ?Y), m(?Y, r_2, ?X). \\ SP(A.r \longleftarrow B_1.r_1 \cap B_2.r_2) &= m(A, r, ?X) :- m(B_1, r_1, ?X), m(B_2, r_2, ?X). \end{aligned}$$

SP extends to a set of statements as expected: $SP(\mathcal{C}) = \{SP(c) \mid c \in \mathcal{C}\}$. Finally, given a policy $\mathcal{C}$, the semantics of a role $A.r \in \mathcal{C}$ is defined in terms of atoms entailed by the semantic program.

Definition 1 (Semantics of a Role). *Let $\mathcal{C}$ be an RT_0 policy, and let $SP(\mathcal{C})$ be the corresponding semantic program. The semantics of a role is defined as follows:*

$$[\![A.r]\!]_{SP(\mathcal{C})} = \{D \mid SP(\mathcal{C}) \models m(A, r, D)\}.$$

2.3 Examples

We now present some examples presenting how RT_0 can be used in different application areas. We begin with an example from [25], showing a typical scenario from the area of electronic commerce.

Example 1. $EPub$ is an electronic publishing company that offers a special discount to anyone who is both a preferred customer of the sister organization, *EOrg*, and an *ACM* member. *Alice* is both. We have the following set $\mathcal{C}$ of credentials:

$$\textit{EPub.spdiscount} \longleftarrow \textit{EOrg.preferred} \cap \textit{ACM.member} \quad (1)$$

$$\textit{EOrg.preferred} \longleftarrow \textit{EOrg.university.student} \quad (2)$$

$$\textit{EOrg.university} \longleftarrow \textit{ABU.accredited} \quad (3)$$

$$\textit{ABU.accredited} \longleftarrow \textit{StateU} \quad (4)$$

$$\textit{StateU.student} \longleftarrow \textit{RegistrarB.student} \quad (5)$$

$$\textit{RegistrarB.student} \longleftarrow \textit{Alice} \quad (6)$$

$$\textit{ACM.member} \longleftarrow \textit{Alice} \quad (7)$$

$$\textit{ACM.member} \longleftarrow \textit{Bob} \quad (8)$$

The semantic program, $SP(\mathcal{C})$, corresponding to the above policy is:

$$m(\textit{EPub}, \textit{spdiscount}, ?X) :- m(\textit{EOrg}, \textit{preferred}, ?X), m(\textit{ACM}, \textit{member}, ?X). \quad (1)$$

$$m(\textit{EOrg}, \textit{preferred}, ?X) :- m(\textit{EOrg}, \textit{university}, ?Y), m(?Y, \textit{student}, ?X). \quad (2)$$

$$m(\textit{EOrg}, \textit{university}, ?X) :- m(\textit{ABU}, \textit{accredited}, ?X). \quad (3)$$

$$m(\textit{ABU}, \textit{accredited}, \textit{StateU}). \quad (4)$$

$$m(\mathit{StateU}, \mathit{student}, ?X) :- m(\mathit{RegistrarB}, \mathit{student}, ?X). \quad (5)$$
$$m(\mathit{RegistrarB}, \mathit{student}, \mathit{Alice}). \quad (6)$$
$$m(\mathit{ACM}, \mathit{member}, \mathit{Alice}). \quad (7)$$
$$m(\mathit{ACM}, \mathit{member}, \mathit{Bob}). \quad (8)$$

The semantics of the roles defined by the set of credentials above is then the following:

$$
\begin{aligned}
[\![\mathit{EPub.spdiscount}]\!]_{SP(\mathcal{C})} &= \{\mathit{Alice}\} \\
[\![\mathit{EOrg.preferred}]\!]_{SP(\mathcal{C})} &= \{\mathit{Alice}\} \\
[\![\mathit{ACM.member}]\!]_{SP(\mathcal{C})} &= \{\mathit{Alice, Bob}\} \\
[\![\mathit{EOrg.university}]\!]_{SP(\mathcal{C})} &= \{\mathit{StateU}\} \\
[\![\mathit{ABU.accredited}]\!]_{SP(\mathcal{C})} &= \{\mathit{StateU}\} \\
[\![\mathit{StateU.student}]\!]_{SP(\mathcal{C})} &= \{\mathit{Alice}\} \\
[\![\mathit{RegistrarB.student}]\!]_{SP(\mathcal{C})} &= \{\mathit{Alice}\}
\end{aligned}
$$

We see then that only *Alice* is eligible for a discount as *Bob*, though being a member of *ACM*, is not a student of an accredited university.

The next example presents the use of RT_0 in collaborating organizations. This example originally appeared in [15].

Example 2. Consider the situation in which two companies: *CITA* (in Italy) and *CUS* (in the US), work on a joint project. *CITA* and *CUS*, have different management structures:

$$
\begin{array}{ll}
\mathit{CITA.partner} \longleftarrow \mathit{Antonio} & \mathit{CUS.ceo} \longleftarrow \mathit{Bob} \\
\mathit{CITA.manager} \longleftarrow \mathit{Luca} & \mathit{CUS.employee} \longleftarrow \mathit{John} \\
\mathit{CITA.programmer} \longleftarrow \mathit{Sandro} & \mathit{CUS.employee} \longleftarrow \mathit{David} \\
\mathit{CITA.all} \longleftarrow \mathit{CITA.partner} & \mathit{CUS.all} \longleftarrow \mathit{CUS.ceo} \\
\mathit{CITA.all} \longleftarrow \mathit{CITA.manager} & \mathit{CUS.all} \longleftarrow \mathit{CUS.employee} \\
\mathit{CITA.all} \longleftarrow \mathit{CITA.programmer} &
\end{array}
$$

In both companies there is an agreement that employees may trust all the sources that are trusted by the *partner* (resp. *ceo*). They can – of course – trust other sources as well.

$$
\begin{array}{ll}
\mathit{Luca.partner} \longleftarrow \mathit{CITA.partner} & \mathit{John.ceo} \longleftarrow \mathit{CUS.ceo} \\
\mathit{Luca.trusted} \longleftarrow \mathit{Luca.partner.trusted} & \mathit{John.trusted} \longleftarrow \mathit{John.ceo.trusted} \\
\mathit{Sandro.partner} \longleftarrow \mathit{CITA.partner} & \mathit{David.ceo} \longleftarrow \mathit{CUS.ceo} \\
\mathit{Sandro.trusted} \longleftarrow \mathit{Sandro.partner.trusted} & \mathit{David.trusted} \longleftarrow \mathit{David.ceo.trusted}
\end{array}
$$

CITA and *CUS* decide to join forces on *projX*, and they agree that most of the documents developed in *projX* should be accessible only to people working on the project, and that some particularly confidential documents should circulate only among the senior personnel. To implement this, the two companies agree to employ the role names *projX* and *seniorprojX*. In *CITA*, the partner decides who participates in projectX, and decides

(in agreement with *CUS*) that the managers of *CITA* should be considered senior people, while in *CUS*, the ceo delegates to *John* the definition of the projectX team as well as of the senior people in it. Finally, *CITA* and *CUS* trust each other's definitions of (senior) people working on projectX. This policy is described and implemented by the following set of credentials.

$$\begin{aligned}
&\mathit{CITA.projX} \longleftarrow \mathit{Antonio.projX}\\
&\mathit{CITA.seniorprojX} \longleftarrow \mathit{CITA.partner}\\
&\mathit{CITA.seniorprojX} \longleftarrow \mathit{CITA.projX} \cap \mathit{CITA.manager}\\
&\mathit{Antonio.projX} \longleftarrow \mathit{Luca}\\
&\mathit{Antonio.projX} \longleftarrow \mathit{Sandro}\\
&\mathit{CITA.projX} \longleftarrow \mathit{CUS.projX}\\
&\mathit{CITA.seniorprojX} \longleftarrow \mathit{CUS.seniorprojX}\\
&\mathit{CUS.projX} \longleftarrow \mathit{John.projX}\\
&\mathit{CUS.seniorprojX} \longleftarrow \mathit{CUS.ceo}\\
&\mathit{CUS.seniorprojX} \longleftarrow \mathit{John.seniorprojX}\\
&\mathit{John.seniorprojX} \longleftarrow \mathit{John}\\
&\mathit{John.projX} \longleftarrow \mathit{John}\\
&\mathit{John.projX} \longleftarrow \mathit{David}\\
&\mathit{CUS.projX} \longleftarrow \mathit{CITA.projX}\\
&\mathit{CUS.seniorprojX} \longleftarrow \mathit{CITA.seniorprojX}
\end{aligned}$$

The following two examples were initially presented by Winsborough and Li in [32]. The first of them shows an example of a co-operation between banking institutions and universities when providing financial support for students. Then, we show an example of policies that can be used by medical suppliers and charity organizations when handling natural disasters.

Example 3. A bank wants to know whether an entity is a full time student in order to determine whether the entity is eligible to defer repayment on a guaranteed student loan (GLS). (The US government insures banks against default of GLSs and requires participating banks to allow full-time students to defer repayments.) The *StateU* university may define its full-time student attribute by the following two credentials:

$$\begin{aligned}
&\mathit{StateU.fullTimeStudent} \longleftarrow \mathit{RegistrarB.fullTimeStudent}\\
&\mathit{StateU.fullTimeStudent} \longleftarrow \mathit{StateU.phdCandidate} \cap \mathit{RegistrarB.partTimeStudent}
\end{aligned}$$

We see that *StateU* says that one is a full-time student if either *RegistrarB* says so, or if one is registered as a Ph.D. candidate at *StateU* and considered part-time student by *RegistrarB*. The following credentials, together with the above ones, show that Bob is a full-time student, i.e. $\mathit{Bob} \in [\![\mathit{StateU.fullTimeStudent}]\!]_{SP(\mathcal{C})}$:

$$\begin{aligned}
&\mathit{StateU.phdCandidate} \longleftarrow \mathit{StateU.gradOfficer.phdCandidate}\\
&\mathit{StateU.gradOfficer} \longleftarrow \mathit{Carol}
\end{aligned}$$

$$Carol.phdCandidate \longleftarrow Bob$$
$$RegistrarB.partTimeStudent \longleftarrow Bob$$

Now, assume that *StateU* is certified by accreditation board *ABU*.

$$ABU.accredited \longleftarrow StateU$$

If universities define *fullTimeStudent* appropriately (for example, as done by *StateU* above), *BankWon* can issue credentials like those bellow to grant loan-deferment permission (denoted by *BankWon.deferGLS*) to students like Bob.

$$BankWon.deferGLS \longleftarrow BankWon.university.fullTimeStudent$$
$$BankWon.university \longleftarrow ABU.accredited$$

Clearly, $Bob \in [\![BankWon.deferGLS]\!]_{SP(\mathcal{C})}$.

Example 4. In the aftermath of a large natural disaster, *MedSup*, a medical supply merchant, offers to sell at a discount medical supplies to be used in the official clean up, which is being organized by a coalition called *ReliefNet*. *Alice* works for *MedixFund*, one of several charity organizations that use private contributions to obtain emergency medical supplies for emergency teams working at the disaster site. The following four credentials show that *Alice* is authorized for the discount.

$$MedixFund.pA \longleftarrow Alice \tag{1}$$
$$ReliefNet.coaMember \longleftarrow MedixFund \tag{2}$$
$$MedSup.partner \longleftarrow ReliefNet.coaMember \tag{3}$$
$$MedSup.discount \longleftarrow MedSup.partner.pA \tag{4}$$

Prior to joining the coalition, *MedixFund* issued credential (1), which states that *Alice* is a purchasing agent for the fund. One of *ReliefNet*'s responsibilities is to identify coalition-member organizations, as it does in credential (2). *MedSup* recognizes these organizations as its coalition partners, as in credential (3), and offers discounted sales to the purchasing agents of those partners, as stated in credential (4). In this example, the judgments of *MedixFund*, *ReliefNet*, and *MedSup* are combined to authorize *Alice*'s receiving a discount from *MedSup*. When *MedSup* enters into other coalition, it can add an additional credential defining *MedSup.partner* to give the discount to the purchasing agents of its new partners.

With the increasing popularity of the P2P networks and their excellent support for sharing of private content, a high demand for flexible user-oriented policies can be observed. Below, we show an example of how RT_0 facilitates the use of personal policies in heterogeneous P2P environment.

Example 5. Charles wants to share his pictures using a P2P file sharing system. He restricts the access to his gallery to his friends and friends of his friends. For his movie collection, Charles applies a somewhat stronger policy: to access it, one has to be a

member of Charles's *friend* role, and a member of the film club Charles is also a member of. The set of credentials, $\mathcal{C}$, modelling this scenario is shown below:

$$\begin{aligned}
&\mathit{Charles.accessMovies} \longleftarrow \mathit{Charles.friend} \cap \mathit{Charles.filmClub}\\
&\mathit{Charles.accessPictures} \longleftarrow \mathit{Charles.friend}\\
&\mathit{Charles.friend} \longleftarrow \mathit{Charles.friend.friend}\\
&\mathit{Charles.friend} \longleftarrow \mathit{Alice}\\
&\mathit{Charles.friend} \longleftarrow \mathit{Bob}\\
&\mathit{Charles.filmClub} \longleftarrow \mathit{Johan}\\
&\mathit{Alice.friend} \longleftarrow \mathit{Jeffrey}\\
&\mathit{Bob.friend} \longleftarrow \mathit{Johan}\\
&\mathit{Johan.friend} \longleftarrow \mathit{Sandro}
\end{aligned}$$

Notice that the delegation depth in RT_0 is unlimited. It means that Charles's role *friend* contains not only friends of his friends, but also friends of friends of his friends and so on (*friends* is a transitive closure of the set of Charles's friends). Therefore, for the given set of credentials, we have the following semantics:

$$\begin{aligned}
[\![\mathit{Charles.accessMovies}]\!]_{SP(\mathcal{C})} &= \{\mathit{Johan}\}\\
[\![\mathit{Charles.accessPictures}]\!]_{SP(\mathcal{C})} &= \{\mathit{Alice, Bob, Jeffrey, Johan, Sandro}\}
\end{aligned}$$

3 $\mathbf{RT_0}$: The Credential Chain Discovery Algorithm

We have seen how RT_0 can be used to define roles and how roles can represent permissions or attributes. We now illustrate the mechanisms needed to answer the *queries* in the RT system. To set the stage, let us first enumerate the three *sorts of queries* we need to cope with. Let $\mathcal{C}$ be a set of credentials.

Sort 1. Given a role $A.r$ and an entity D, determine whether $D \in [\![A.r]\!]_{SP(\mathcal{C})}$.
Sort 2. Given a role $A.r$, determine its member set, $[\![A.r]\!]_{SP(\mathcal{C})}$.
Sort 3. Given an entity D, determine all the roles it is a member of, i.e. generate the set $\{A.r \mid D \in [\![A.r]\!]_{SP(\mathcal{C})}\}$.

Notice that while queries of Sort 1 simply require a yes/no answer, the other two sorts require to generate a whole set. Also, notice that queries of Sort 2 and 3 are strictly more expressive than queries of Sort 1: if we are able to answer a query of Sort 2 or 3 we are certainly able to answer a query of Sort 1, while the opposite is not true. At this stage, one might wonder if Sort 3 queries are actually needed. This will become clear in the sequel.

Remark 1. Technically, this section is based on [25] with the additional simplifying assumption that queries may refer only to roles and principals (and not to role expressions, e.g. we do not allow queries such as "given a role expression $A.r_1.r_2$, determine its member set $[\![A.r_1.r_2]\!]_{SP(\mathcal{C})}$"). This assumption allows us to simplify the notation by a great deal, and does not limit the expressiveness of the framework, as one can always introduce a new role to take the meaning of a role expression.

The algorithms we present in this section operate on a *credential graph*, which is a directed graph representing a set $\mathcal{C}$ of credentials and is built as follows: each node $[e]$ represents a role expression e; every credential $A.r \longleftarrow e$ in $\mathcal{C}$ contributes to the graph an edge from $[e]$ (the node representing e) to $[A.r]$ (the node representing $A.r$), which is denoted by $[A.r] \Leftarrow [e]$, and is called a credential edge. A path in the graph from the node $[e_1]$ to the node $[e_2]$ consists of zero or more edges and is denoted $[e_2] \stackrel{*}{\Leftarrow} [e_1]$. Additional edges, called *derived edges*, are added to handle linked roles and intersections. These edges are called derived edges because their inclusion in the credential graph comes from the existence of other, semantically related, paths in the graph.

Given a set $\mathcal{C}$ of credentials, we define the following finite structures: Entities($\mathcal{C}$) is the set of entities in $\mathcal{C}$, Names($\mathcal{C}$) is the set of role names in $\mathcal{C}$, and RoleExpressions($\mathcal{C}$) is the set of role expressions that can be constructed using Entities($\mathcal{C}$) and Names($\mathcal{C}$), i.e.:

$$\text{RoleExpressions}(\mathcal{C}) = \begin{cases} A, & \\ A.r_1, & \text{where } A, B_1, B_2 \in \text{Entities}(\mathcal{C}), \\ A.r_1.r_2, & \phantom{\text{where }} r_1, r_2 \in \text{Names}(\mathcal{C}) \\ B_1.r_1 \cap B_2.r_2 & \end{cases}$$

The following definition is a simplified version of Definition 2 in [25] (see Remark 1). Thanks to this simplification we can restrict our attention to the *basic* credential graph and avoid some complexities from the original presentation.

Definition 2 (Basic Credential Graphs). *Let $\mathcal{C}$ be a set of RT$_0$ credentials. The* basic credential graph *$G_\mathcal{C}$ relative to $\mathcal{C}$ is defined as follows: the set of nodes $N_\mathcal{C}$ = RoleExpressions($\mathcal{C}$) and the set of edges $E_\mathcal{C}$ is the least set of edges over $N_\mathcal{C}$ that satisfies the following three closure properties:*

- *Closure property 1: If $A.r \longleftarrow e \in \mathcal{C}$, then $[A.r] \Leftarrow [e] \in E_\mathcal{C}$. $[A.r] \Leftarrow [e]$ is called a* credential edge.
- *Closure property 2: If there exists a path $[A.r_1] \stackrel{*}{\Leftarrow} [B]$ in $G_\mathcal{C}$, then $[A.r_1.r_2] \Leftarrow [B.r_2] \in E_\mathcal{C}$. We call $[A.r_1.r_2] \Leftarrow [B.r_2]$ a derived link edge, and call the path $[A.r_1] \stackrel{*}{\Leftarrow} [B]$ a* support set *for this edge.*
- *Closure property 3: If $D, B_1.r_1 \cap B_2.r_2 \in N_\mathcal{C}$, and there exist paths $[B_1.r_1] \stackrel{*}{\Leftarrow} [D]$ and $[B_2.r_2] \stackrel{*}{\Leftarrow} [D]$ in $G_\mathcal{C}$, then $[B_1.r_1 \cap B_2.r_2] \Leftarrow [D] \in E_\mathcal{C}$. This is called a* derived intersection edge*, and $\{[B_1.r_1] \stackrel{*}{\Leftarrow} [D], [B_2.r_2] \stackrel{*}{\Leftarrow} [D]\}$ is a* support set *for this edge.*

The set of edges $E_\mathcal{C}$ can be constructed inductively as follows. We start with the set $E_\mathcal{C}^0 = \{[A.r] \Leftarrow [e] \mid A.r \longleftarrow e \in \mathcal{C}\}$ and then construct $E_\mathcal{C}^{i+1}$ from $E_\mathcal{C}^i$ by adding one edge according to either closure property 2 or 3. Since $N_\mathcal{C}$ is finite, the order in which edges are added is not important, and the sequence $\{E_\mathcal{C}^i\}_{i \in \mathcal{N}}$ converges to $E_\mathcal{C}$.

Example 6. Figure 1 shows a subset of the basic credential graph for the set of credentials in Example 1. Edges labelled with numbers are credential edges, and the numbers correspond to the ones marking credentials in Example 1. The two edges without labels are derived edges: one added by the closure property 2 ([*EOrg.university.student*] $\Leftarrow$ [*StateU.student*]), and one by the closure property 3 ([*EOrg.preferred* $\cap$ *ACM.member*] $\Leftarrow$ [*Alice*]).

In [25] it is proven that the credential graphs are sound and complete w.r.t. to the set-theoretic semantics: if there is a path $[e_2] \overset{*}{\Leftarrow} [e_1]$ in any $G_{\mathcal{C}}$, then $expr[\mathcal{S}_{\mathcal{C}}](e_2) \supseteq expr[\mathcal{S}_{\mathcal{C}}](e_1)$, and if $D \in expr[\mathcal{S}_{\mathcal{C}}](e_0)$, then there exists path $[e_0] \overset{*}{\Leftarrow} [D]$ in $G_{\mathcal{C}}$. Here $expr[\mathcal{S}_{\mathcal{C}}](e)$ is the set-theoretic semantics of a role expression e, which can be proven in a straightforward way to be equivalent to the LP based semantics we have introduced in Sect. 2.2.

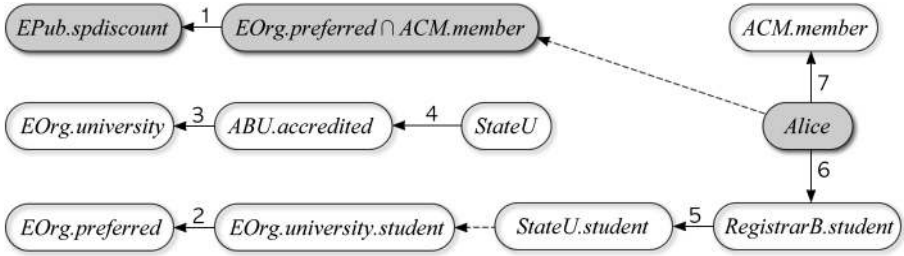


Fig. 1. The subset of the credential graph for the set of credentials in Example 1 containing path from *EPub.spdiscount* to *Alice*

Therefore, given a set $\mathcal{C}$ of credentials, we can answer each of the queries enumerated at the beginning of this section by consulting a basic credential graph of $\mathcal{C}$. Constructing the path $[A.r] \overset{*}{\Leftarrow} [D]$ alone proves that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$, provided that each derived edge has at least one support set. The portion of the credential graph that must be constructed for it is what we call a *credential chain*.

Definition 3 (Credential Chains). *Given a set $\mathcal{C}$ of credentials, a role $A.r$, and an entity D, a* credential chain *from D to $A.r$, denoted $\langle A.r \twoheadleftarrow D\rangle$, is a minimal subset of $E_{\mathcal{C}}$ containing a path $[A.r] \overset{*}{\Leftarrow} [D]$ and also containing a support set for each derived edge in the subset.*

The chain discovery starts at the node representing the requester, or at the node representing the role (permission) to be proven, or both, and then traversing paths in the graph trying to build an appropriate chain. In addition to being goal-directed, this approach allows the elaboration of the graph to be scheduled flexibly. Also, the graphical representation of the evaluation state makes it relatively straightforward to manage cyclic dependencies.

In the rest of this section we illustrate the three algorithms originally defined in [25] to answer the three sorts of queries, listed at the top of this section (with the simplifying assumption illustrated in Remark 1). The backward search algorithm (also called the top-down algorithm) (Sect. 3.1) answers the second sort of queries, i.e. it determines all members of a role expression. The forward search algorithm (also called the bottom-up algorithm) in Sect. 3.2 answers the third sort of queries, i.e. it determines all roles that an entity is a member of. The bidirectional search algorithm (Sect. 3.3) answers the first sort of queries, i.e. it determines whether an entity is a member of a role expression. Note that in this section we assume that credentials are stored in such a way that we can

list them all at any time. In practice, this is not always the case. We address the problem of distributed storage in the next section.

3.1 The Backward Search Algorithm

The backward search algorithm can determine all the members of a given role $A.r$. In terms of the credential graph, it finds all the entity nodes that can reach the node $A.r$, and for each such entity D, it constructs a chain $\langle A.r \twoheadleftarrow D \rangle$. It is called backward because it follows edges in the reverse direction. The algorithm works by constructing a *proof graph*, which is a data structure that represents a credential graph and maintains certain information on the nodes. Listing 1.1 shows the algorithm in pseudo-code using a Python-like syntax. We have four classes: *ProofGraph* representing the proof graph, *ProofNode* representing proof graph nodes, *BLinkingMonitor* and *BIntersectionMonitor* used to handle linked and intersection roles respectively.

The ProofGraph class stores the set of nodes and the set of edges corresponding to the set of nodes and the set of edges in the basic credential graph in its instance variables: *nodes* and *edges* respectively. Adding nodes and edges is handled by the *addNode()* and *addEdge()* methods. The main processing is handled by the *bProcess()* method of the ProofGraph class. The nodes to be processed are stored in the backward processing queue (*bQueue*).

Each node in the graph is represented by an instance of the ProofNode class. Each ProofNode object stores the set of backward solutions in the *bSolutions* attribute. A *solution* in the backward search algorithm is an entity. Thus, the *solution* attribute of the ProofNode class stores all the entities which are known to be members of the corresponding role expression. When a new solution D is discovered, every node e such that there is a path $[e] \stackrel{*}{\Leftarrow} [D]$ in the proof graph must be notified about this solution. This is realized using a well-know *observer* design pattern. Every instance of the ProofNode class maintains a list of observers, called *backward solution monitors* in the text. When a node is notified about one or more new solutions – by invoking node's *notify()* operation – it immediately notifies all the monitors (observers) of the node using node's *notifyAll()* operation. Every node which is not an entity node (entities do not have any solutions other than themselves) can be *registered* as a backward monitor of a node using node's *bAttach()* operation. There are two special backward monitors that are not instances of the ProofNode class: backward linking and intersection monitors. In Listing 1.1 they are represented by two classes: *BLinkingMonitor* and *BIntersection-Monitor*. Linking and intersection monitors realize the basic credential graph closure properties 2 and 3 respectively.

When processing a linked role $A.r_1.r_2$, the algorithm first creates a new node for the role $A.r_1$, then it creates a backward linking monitor and attaches this monitor to $[A.r_1]$. The backward linking monitor works as follows: when the backward linking monitor corresponding to a linked role $A.r_1.r_2$ is notified about a new solution B, it means that B became a member of $A.r_1$. By the closure property 2 in Definition 2 this implies that the basic credential graph contains the edge $[A.r_1.r_2] \Leftarrow [B.r_2]$. The backward linking monitor realizes this by creating new node corresponding to role $B.r_2$ and by adding the edge $[A.r_1.r_2] \Leftarrow [B.r_2]$ to the proof graph (lines (42–45)).

When processing an intersection node $[B_1.r_1 \cap B_2.r_2]$, the algorithm first creates two new nodes $[B_1.r_1]$ and $[B_2.r_2]$, then it creates a backward intersection monitor and attaches this monitor to these two newly created nodes. When any of these two nodes receives a new solution D, it notifies all of its backward solution monitors, including the monitor associated to $B_1.r_1 \cap B_2.r_2$. When this monitor is notified about solution D it checks how many times it observed the addition of entity D. When the counter reaches 2, it adds edge $[B_1.r_1 \cap B_2.r_2] \Leftarrow [D]$ to the proof graph (lines (47–51)).

In order to find all members of a role $A.r$ the algorithm is initialized using the following sequence:

```
proofGraph = new ProofGraph()
proofGraph.addNode(A.r)
proofGraph.bProcess()
```

Listing 1.1. Backward Search Algorithm

```
1   class ProofGraph:
      def bProcess():
3       while not bQueue.empty():
          n = bQueue.dequeue()
5         if n is an entity D:
            n.bSolutions.add(D)
7           n.notifyAll (D)
            continue
9         if n is a role A.r:
            foreach A.r ⟵ e ∈ C:
11            addNode(e)
              addEdge([A.r] ⇐ [e])
13          continue
          if n is a linked role A.r1.r2:
15          n1 = addNode(A.r1)
            n1.bAttach(new BLinkingMonitor(A.r1.r2))
17          continue
          if n is an intersection B1.r1 ∩ B2.r2:
19          n1 = addNode(B1.r1)
            n2 = addNode(B2.r2)
21          m = new BIntersectionMonitor(B1.r1 ∩ B2.r2)
            n1.bAttach(m)
23          n2.bAttach(m)
            continue
25    def addNode(e):
        if nodes.contains(e): return getNode(e)
27      n = new ProofNode(e)
        nodes.add(n)
29      bQueue.enqueue(n)
        return n
```

```
31      def addEdge([e2] ⇐ [e1]):
          n1 = getNode(e1)
33        n2 = getNode(e2)
          if not edges.contains([n2] ⇐ [n1]):
35          edges.add([n2] ⇐ [n1])
          if n1.hasSolutions():
37          s = n1.getSolutions()
            n2.bSolutions.add(s)
39          n2.notifyAll(s)
          n1.bAttach(n2)
41
    class BLinkingMonitor(A.r1.r2):
43    def notify(B):
        n = proofGraph.addNode(B.r2)
45      proofGraph.addEdge([A.r1.r2] ⇐ [B.r2])

47  class BIntersectionMonitor(B1.r1 ∩ B2.r2):(A.r1.r2):
      def notify(D):
49      solutions.add(D)
        if solutions.count(D) == 2:
51        proofGraph.addEdge([B1.r1 ∩ B2.r2] ⇐ [D])

53  class ProofNode:
      def bAttach(m):
55      bMonitors.add(m)
      def notify(solutions):
57      bSolutions.add(solutions)
        notifyAll(solutions)
59    def notifyAll(solutions):
        foreach m in bMonitors:
61        m.notify(solutions)
```

Example 7. Figures 2(a)-(d) illustrate the process of constructing the proof graph by doing backward search from *EPub.discount* for the following set of credentials $\mathcal{C}$ (a subset of Example 1). This corresponds to the query of Sort 1: determine the set of members of *EPub.spdiscount*, $[\![EPub.spdiscount]\!]_{SP(\mathcal{C})}$.

$$EPub.spdiscount \longleftarrow EOrg.preferred \cap ACM.member \tag{1}$$

$$EOrg.preferred \longleftarrow EOrg.university.student \tag{2}$$

$$EOrg.university \longleftarrow ABU.accredited \tag{3}$$

$$ABU.accredited \longleftarrow StateU \tag{4}$$

$$StateU.student \longleftarrow RegistrarB.student \tag{5}$$

$$RegistrarB.student \longleftarrow Alice \tag{6}$$

$$ACM.member \longleftarrow Alice \tag{7}$$

In Figs. 2(a)-(d), the first line of each node gives the node number (following the order of creation) and the role expression represented by the node. The second line lists the solutions associated to the node. To simplify the reading, we have labelled each solution and each graph edge with the number of the node that was being processed when the solution or edge was added. In each of the figures dashed edges and nodes are the newly processed nodes while the newly added solutions are grey. Below we repeat the process of the construction of this proof graph.

The algorithm starts the search from *EPub.spdiscount*. The only credential defining role *EPub.spdiscount* is (1). To process it, the algorithm adds the new node [*EOrg.preferred* $\cap$ *ACM.member*] to the proof graph, and it inserts it in the queue of nodes *bQueue* (lines (25–30)). Then the algorithm adds a credential edge from the newly added node to *EPub.spdiscount* (Fig. 2(a)). We label the edge with number 0 to indicate that this edge was added while processing node *EPub.spdiscount*. The new node is an intersection. To process it, the algorithm first creates two new nodes: [*EOrg.preferred*] and [*ACM.member*], and adds them to the processing queue in this order. Next it creates an intersection monitor and it attaches it to both [*EOrg.preferred*] and [*ACM.member*] (lines (18–24, and the two edges labelled with 1 in Fig. 2(a)). This monitor guarantees that if the same solution D appears in both [*EOrg.preferred*] and [*ACM.member*], a derived edge is added from [D] to [*EOrg.preferred* $\cap$ *ACM.member*] (lines (47–51)). The next node to process is [*EOrg.preferred*]. The only credential defining this role is the linking inclusion *EOrg.preferred* $\longleftarrow$ *EOrg.university.student*. The algorithm adds node [*EOrg.university.student*] to the graph, and a credential edge from this node to [*EOrg.preferred*] (Fig. 2(b)). Next, the node [*ACM.member*] is processed. Giving the presence of the credential *ACM.member* $\longleftarrow$ *Alice*, the algorithm adds a new node [*Alice*] to the graph and to the processing queue. This node will be processed *after* the node [*EOrg.university.student*], so we do not add any solution at this stage.

The next node to process is [*EOrg.university.student*]. As this is a node representing a linked role, the algorithm first adds new node [*EOrg.university*] to the proof graph (and also to the processing queue) and then it attaches a linking monitor to [*EOrg.university*] (lines (14–17 and Edge 4 in Fig. 2(b)). This monitor behaves as follows: each time [*EOrg.university*] receives a new solution B, it creates a node for *B.student* and adds the derived edge from [*B.student*] to [*EOrg.university.student*] (lines (42–45)).

The next node to process is [*Alice*]. As this is an entity node, it immediately receives *Alice* as solution and it notifies all its backward solution monitors: in our case [*ACM.member*]. The intersection monitor stored by [*ACM.member*] observes that $Alice$ is the received solution, but takes no action as it has been added only to [*ACM.member*], and does not appear as a solution at *EOrg.preferred* yet.

In a similar manner, [*EOrg.university*] receives the solution $StateU$ when processing node [$StateU$] (Fig. 2(c)). After this, the linking monitor stored at [*EOrg.university*] creates the new node [*StateU.student*].

When *StateU.student* receives the solution $Alice$ from [*RegistrarB.student*] (Fig. 2(d)), this solution is propagated upward to [*EOrg.university.student*] and [*EOrg.preferred*]. The intersection monitor at node [*EOrg.preferred*] observes that $Alice$ is added for the second time, this time by means of node [*EOrg.preferred*], and

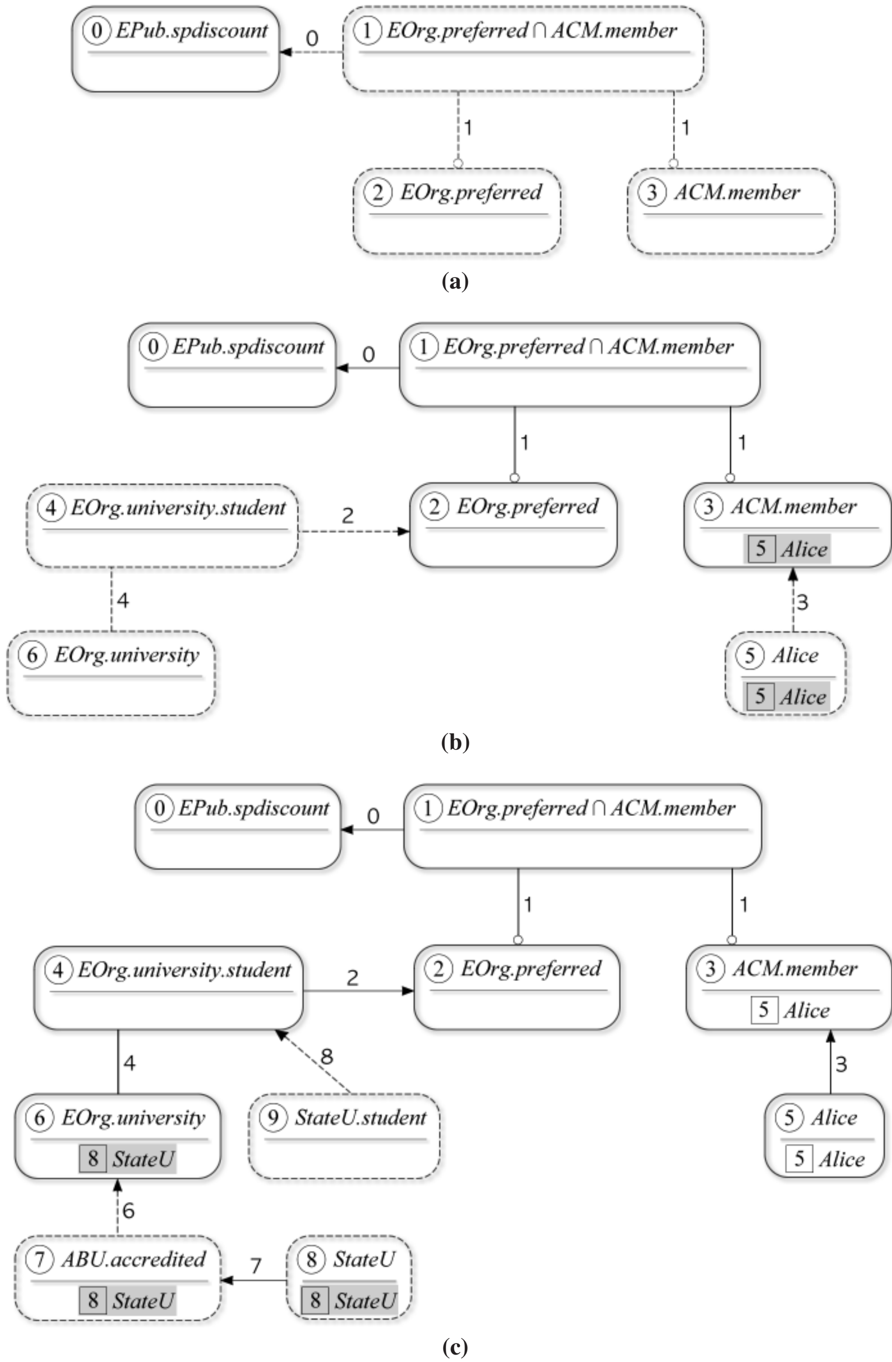
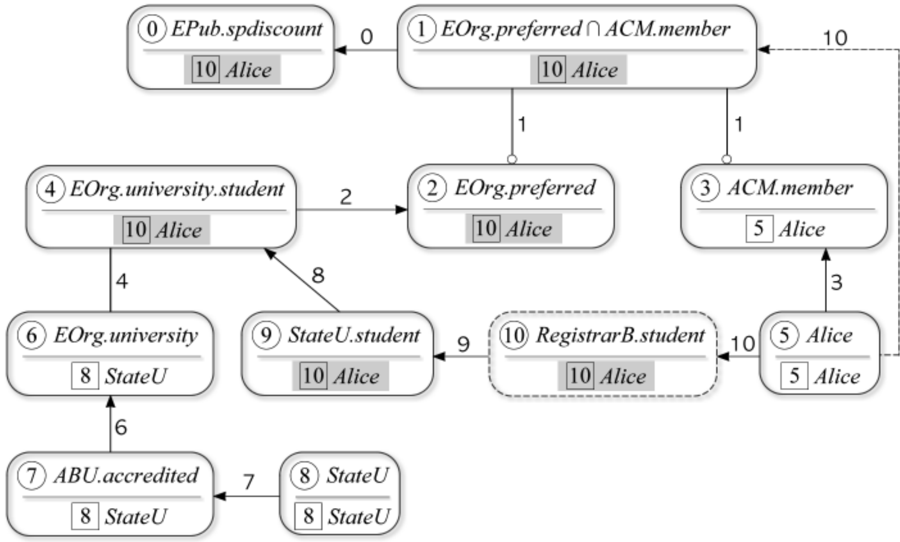


Fig. 2. Backward search from *EPub.spdiscount*



(d)

Fig. 2. (*continued*)

in response it creates a derived edge from $[Alice]$ to [*EOrg.preferred* ∩ *ACM.member*]. The solution $Alice$ is then immediately copied from $[Alice]$ to node [*EOrg.preferred* ∩ *ACM.member*] and then to [*EPub.spdiscount*] (lines (36–39)).

At this point, there are no more nodes to process and the algorithm terminates. Given the set of credentials shown above, *EPub.spdiscount* has only one member: $Alice$.

3.2 The Forward Search Algorithm

The forward search algorithm answers queries of the third sort, i.e. it finds all roles that contain a given entity D_0 as a member. The direction of the search moves from the subject of a credential towards its issuer.

The forward algorithm has the same overall structure as the backward algorithm. It constructs a proof graph, maintaining a queue of nodes to be processed; both contain initially just one node, $[D_0]$. Nodes are processed one by one until the queue is empty. Listing 1.2 reports the algorithm's pseudo-code.

A solution in the forward search algorithm can be a *full solution* or a so called *partial solution*. A full solution is a role and indicates that the initial node is a member of this role. Partial solutions are necessary to properly handle intersections (see Closure Property 3 in Definition 2). Given an intersection $B_1.r_1 \cap B_2.r_2$ a partial solution has the form $(B_1.r_1 \cap B_2.r_2, i)$ where $i \in \{1, 2\}$. We add the partial solution $(B_1.r_1 \cap B_2.r_2$,i) to the node $[e]$ when $[B_i.r_i]$ is reachable from $[e]$ (lines (8–9)).

Similarly to the backward processing algorithm, when a node receives either a full, or a partial solution, it notifies each of its forward solution monitors. The solutions travel through the edges eventually reaching some other entity node $[D]$. When $[D]$ is notified

about new partial solution $(B_1.r_1 \cap B_2.r_2, i)$, it checks whether it has the two partial solutions $(B_1.r_1 \cap B_2.r_2, 1)$ and $(B_1.r_1 \cap B_2.r_2, 2)$, and, if so, it adds a derived edge $[B_1.r_1 \cap B_2.r_2] \Leftarrow [D]$ to the proof graph (lines (37–37).

Linking roles are handled using forward linking monitors. A linking monitor is created when processing a role $B.r_2$. A new node $[B]$ is created and a forward linking monitor *FLinkingMonitor(*$B.r_2$*)* is attached to $[B]$ (lines (12–13)). This monitor, when notified by $[B]$ about new solution $A.r_1$, creates new node $[A.r_1.r_2]$ and adds it to the proof graph and to the forward processing queue. Then, it adds new edge $[A.r_1.r_2] \Leftarrow [B.r_2]$ to the proof graph (lines (27–30)).

In order to find all roles $A.r$ an entity D_0 is a member of, the algorithm should be initialized using the following sequence:

```
proofGraph = new ProofGraph()
proofGraph.addNode(D)
proofGraph.fProcess ()
```

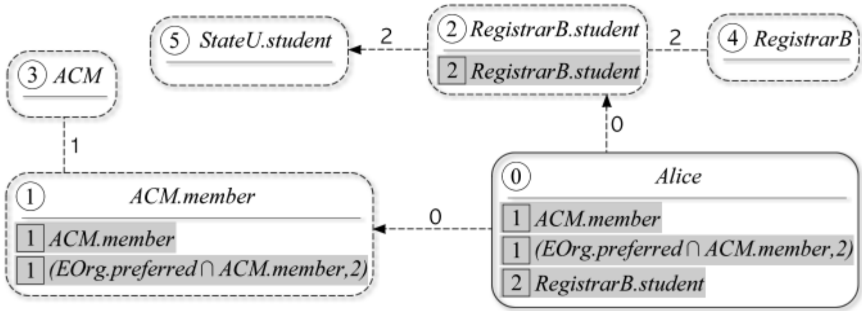
Listing 1.2. Forward Search Algorithm

```
class ProofGraph:
  def fProcess ():
    s = ∅
    while not fQueue.empty():
      n = fQueue.dequeue()
      if n is a role B.r_2:
        s. add(B.r_2)
        foreach A.r ⟵ f_1 ∩ f_2 ∈ C s.t. ∃ i ∈ {1, 2}, f_i = B.r_2:
          s. add(( f_1 ∩ f_2, i))
        n. fSolutions . add(s)
        n. notifyAll (s)
        n_1 = addNode(B)
        n_1.fAttach(new FLinkingMonitor(B.r_2))
      # get the role expression associated with node n
      e = n. roleExpression ()
      foreach A.r ⟵ e ∈ C:
        addNode(A.r)
        addEdge([A.r] ⇐ [e])
  def addNode(e): # see Listing 1.1 line 25 for the definition
  def addEdge([e_2] ⇐ [e_1]):
    n_1 = getNode(e_1)
    n_2 = getNode(e_2)
    if not edges. contains ( [n_2] ⇐ [n_1]):
      edges. add([n_2] ⇐ [n_1])
    n_2.fAttach(n_1)

class FLinkingMonitor(B.r_2):
  def notify (A.r_1):
```

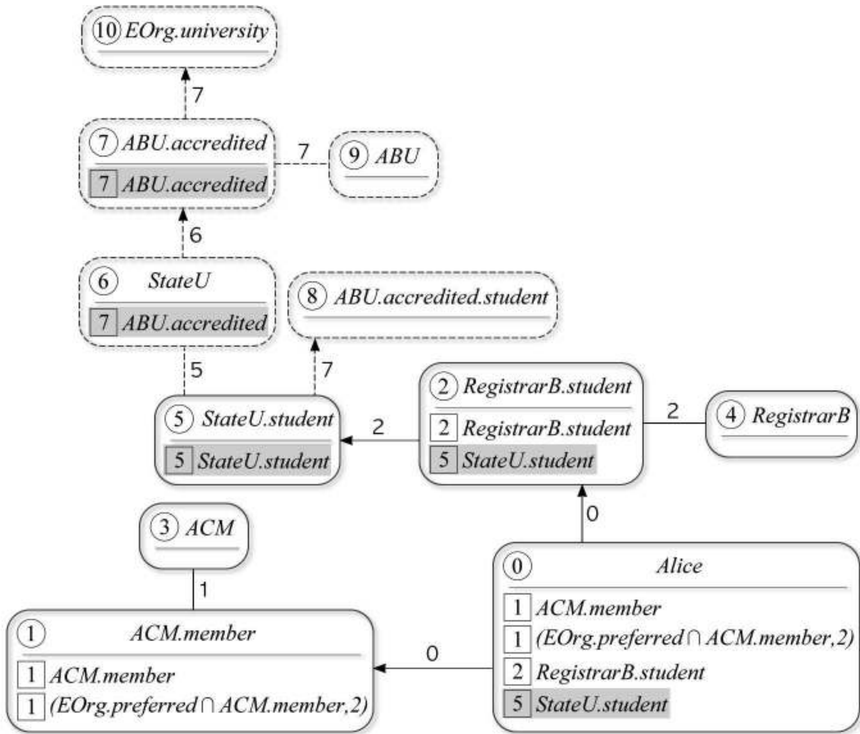
```
    proofGraph.addNode(A.r1.r2)
    proofGraph.addEdge([A.r1.r2] ⇐ [B.r2])

class ProofNode:
  def fAttach(m):
    fMonitors.add(m)
  def notify(solutions):
    fSolutions.add(solutions)
    if the node is an entity node D:
      foreach f1 ∩ f2 s.t. ∀i ∈ {1,2}∃(f1 ∩ f2, i) ∈ fSolutions:
        proofGraph.addNode(f1 ∩ f2)
        proofGraph.addEdge([f1 ∩ f2] ⇐ [D])
    else: notifyAll(solutions)
  def notifyAll(solutions):
    foreach m in fMonitors:
      m.notify(solutions)
```

Example 8. Figures 3(a)-(c) depict the process of constructing the proof graph by forward search from $[Alice]$ for the set of credentials from Example 1.

The first line of each node reports the node number in order of creation and the role expression represented by the node. The second part of a node lists the solutions associated to the node. Each solution and each graph edge is labelled with the number of the node that was being processed when the solution or edge was added. In each of the figures the dashed edges and nodes are the new ones and the new solutions are grayed. The process begins from node [*Alice*] (Fig. 3(a)). As [*Alice*] is an entity node, the algorithm searches for all credentials having *Alice* as the body. There are two such credentials: *ACM.member* ⟵ *Alice* and *RegistrarB.student* ⟵ *Alice*. Thus, the algorithm creates two nodes: [*ACM.member*] and [*RegistrarB.student*] and adds two credential edges from [*Alice*] to them. The next node to be processed is [*ACM.member*] (recall that the number in the circle displays the order of the processing). *ACM.member* is a role. Therefore, the algorithm first adds *ACM.member* as a solution to it. Next, it checks if there are any intersection credentials having *ACM.member* in the body. The role *ACM.member* appears as the second component of *EOrg.preferred* ∩ *ACM.member* in credential (1). Thus, the algorithm adds the partial solution (*EOrg.preferred* ∩ *ACM.member*,2) to the solution space of [*ACM.member*] (lines (8–9)). The node [*ACM.member*] notifies all its forward solution monitors about the new solutions. So, [*Alice*] receives *ACM.member* and (*EOrg.preferred* ∩ *ACM.member*,2) as its first solutions. Now, the algorithm creates the node [*ACM*] and a forward linking monitor (edge with number 1 in Fig. 3(a)), which is then added as a solution monitor to [*ACM*] (lines (12–13)). This monitor, on observing that [*ACM*] gets a full solution $A.r$, creates the node [$A.r$.*member*] and adds the edge from [*ACM.member*] to [$A.r$.*member*] to the proof graph (lines (27–30)). The node [*RegistrarB*] is processed in a similar way. There are no credentials having *ACM* or *RegistrarB* as the body, so they do not have any solutions. Figure 3(a) shows the snapshot of the graph after processing of node *RegistrarB*.



(a)

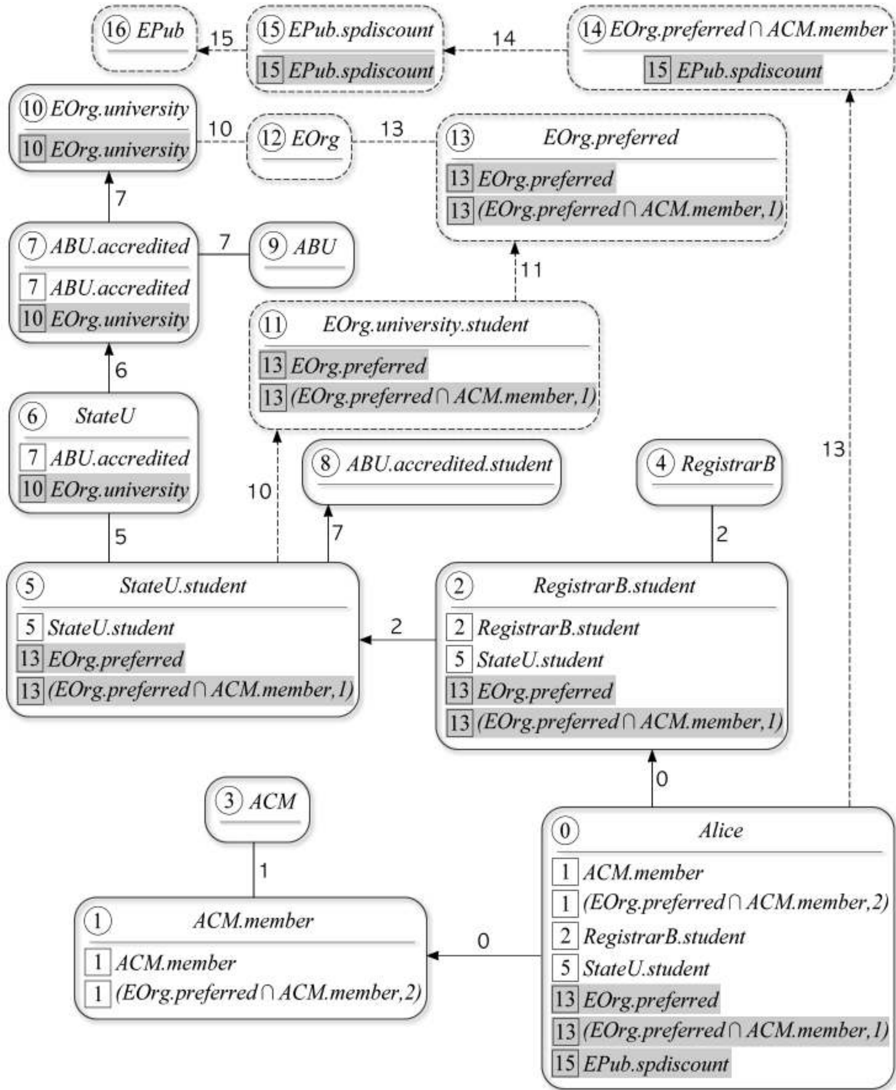


(b)

Fig. 3. Forward search from *Alice*

Figure 3(b) shows the graph after processing of node *ABU*.*accredited*. The nodes [*ABU*] and [*EOrg*.*university*] are the ones added when processing [*ABU*.*accredited*] and are next to be processed. When [*StateU*] receives the solution [*ABU*.*accredited*] its (forward) linking monitor creates node [*ABU*.*accredited*.*student*].

Figure 3(c) shows the complete graph. [*EOrg*.*preferred*] has one full solution, *EOrg*.*preferred*, and one partial solution (*EOrg*.*preferred* ∩ *ACM*.*member*,1), which



(c)

Fig. 3. (*continued*)

comes from the fact that *EOrg.preferred* is the first component of the intersection in the body of credential (1). [*EOrg.preferred*] notifies its forward solution monitors about these two solutions, which eventually reach [*Alice*]. When [*Alice*] is notified, since it has the two partial solutions corresponding to the intersection *EOrg.preferred* ∩ *ACM.member*, it creates the intersection node [*EOrg.preferred* ∩ *ACM.member*] and the edge from [*Alice*] to it. Finally, [*Alice*] receives the solution from [*EPub.spdiscount*].

3.3 The Bidirectional Search Algorithm

The two algorithms presented in Sect. 3.1 and Sect. 3.2 can also be used to answer the queries of Sort 1 presented at the top of Sect. 3 in which given a role $A.r$ and an entity D, one wants to determine whether $D \in [\![A.r]\!]_{SP(\mathcal{C})}$. This can be done either by using the backward search and starting from $A.r$ or by using forward search and staring from D. It is also possible to perform both searches at the same time. Such an algorithm is called *bidirectional search algorithm*. This may not make too much sense at first – as the bidirectional search algorithm may construct a larger graph than does either backward or forward search – but as we show later in Sect. 4, this may be very useful when the credential storage is distributed. We leave as an exercise for the reader to merge the two algorithms presented in Sects. 3.1 and 3.2 in order to obtain the bidirectional search algorithm.

4 The Storage Type System

Winsborough and Li argue that a trust management language should have some support for the *distributed* credential storage [32]. In our description so far, we assumed that the credential storage is centralized; more precisely, we have assumed that at any time we can list the whole set of credentials. Such an assumption is not realistic in practice, as sometimes we may want to store the credentials by their issuers and sometimes by their subjects (see [25,32] for a discussion). Intuitively, the problem with decentralized storage is that one may not know where to find the credentials needed to build a proof. Let us see an example of this.

Example 9. Assume that the policy contains only two credentials:

$$A.r \longleftarrow B.r_1 \tag{1}$$

$$B.r_1 \longleftarrow D \tag{2}$$

Now, assume that one wants to know whether $D \in [\![A.r]\!]_{SP(\mathcal{C})}$. Each of these two credentials could be stored at either its issuer and/or its subject.

First, let us assume that credential (1) is stored at A and credential (2) at D. Using backward search, we start from node $[A.r]$ by listing all credentials defining $A.r$. The only credential stored at A is $A.r \longleftarrow B.r_1$, so, the only way to proceed from here is to "go to" B, but since B does not store any credentials, the backward search algorithm concludes that $[\![A.r]\!]_{SP(\mathcal{C})}$ is empty. In the forward search algorithm we would start from $[D]$ by searching for all the credentials having D as the body. D stores only one credential: $B.r_1 \longleftarrow D$. The forward search algorithm then "goes to" B and fetches the credentials it stores. However, since B does not store any credentials, the forward search algorithm concludes that the only role D is a member of is $B.r$. Also, the forward search does not allow us to prove that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$. The bidirectional search algorithm, on the other hand, succeeds because when backward search stops at node $B.r_1$ it knows from the forward search that D is a member of $B.r_1$. Therefore, it can conclude that D must be the member of $A.r$ as well.

Second, and perhaps more importantly, suppose that the two credentials above were stored at entity B (i.e. that (1) was stored by the subject and (2) was stored by the

issuer). In this case, following the same reasoning, it is easy to see that both forward and backward search algorithms fail again, but, in addition, even the bidirectional search fails.

When both credentials are stored by their issuers (i.e. credential (1) is stored at A and credential (2) is stored at B) the only way to discover that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$ is by using backward search starting from $A.r$.

Finally, when both credentials are stored by their subjects (i.e. (1) is stored at B and (2) is stored at D) only the forward search starting from D can find out that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$.

This example shows that when credential storage is distributed some chain discovery algorithms may or may not work. In particular, if credential storage is not regulated, one may be unable to find the answers to a query.

RT_0 deals with this problem by introducing a *storage type system* limiting the number of possible storage location by introducing the notion of *well-typed* credentials. Each role name r has two types: an issuer-side type and a subject-side type. On the issuer side, each role name can have one of three type values: *issuer-traces-none*, *issuer-traces-def*, and *issuer-traces-all*. On the subject side, each role can have one of two type values: *subject-traces-none* and *subject-traces-all*. The intuition behind these type values is the following: if a role name r has the (issuer-side) type issuer-traces-all then one should be able to answer the queries of Sort 2 and to find all members of any role of the form $A.r$ using solely the backward search algorithm. Similarly, if a role name r has (subject-side) type subject-traces-all then starting from any entity D one should be able to find all roles of the form $A.r$ such that D is a member of $A.r$ (which corresponds to the queries of Sort 3). The type value issuer-traces-def is a weaker version of the issuer-traces-all type value. If a role name r has (issuer-side) type issuer-traces-def, then from any entity A one can find all credentials defining $A.r$. If a role name r has type value issuer-traces-none then for any role $A.r$, the backward search algorithm will not find any member of this role. If a role name r has type value subject-traces-none, then starting from any entity D, the forward search algorithm will not be able to find any role $A.r$ such that D is a member of $A.r$.

Summarizing, we have the following definition:

Definition 4 (Type). *A* type *is a mapping from role names into two-element sets of the form* $\{i, s\}$*, such that:*

- $i \in \{$*issuer-traces-all, issuer-traces-def, issuer-traces-none*$\}$*, and*
- $s \in \{$*subject-traces-all, subject-traces-none*$\}$.

We call i the *issuer-side* type value and s the *subject-side* type value of r, denoted $itype(r)$ and $stype(r)$ respectively, and we let $type(r) = itype(r) \cup stype(r)$.

The type of a role name directly indicates the storage location of the credentials.

Definition 5 (Storage). *Let r be a role name and $A.r \longleftarrow e$ be a credential.*

- *If $itype(r) \in \{$issuer-traces-all, issuer-traces-def$\}$ then A must store this credential.*
- *If $stype(r) =$ subject-traces-all then every entity $B \in base(e)$ must store credential $A.r \longleftarrow e$.*

Table 1. Well Typed RT_0 credentials

	$A.r \longleftarrow B.r_1$		
r_1	ITA	ITD	STA
r: ITA	OK		
r: ITD	OK	OK	OK
r: STA			OK

(a)

	$A.r \longleftarrow A.r_1.r_2$								
r_1	ITA			ITD			STA		
r_2	ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
r: ITA	OK								
r: ITD	OK	OK	OK			OK			OK
r: STA									OK

(b)

	$A.r \longleftarrow B_1.r_1 \cap B_2.r_2$								
r_1	ITA			ITD			STA		
r_2	ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
r: ITA	OK	OK	OK	OK			OK		
r: ITD	OK	OK	OK	OK	OK	OK	OK	OK	OK
r: STA			OK			OK	OK	OK	OK

(c)

Notice that a credential might have to be stored both by the issuer and by the subject (this is the case e.g. when one wants to be able to answer the queries of both Sort 2 and Sort 3). The type value issuer-traces-none (resp. subject-traces-none) indicates that A (resp. any entity $B \in \mathit{base}(e)$) does not store credential $A.r \longleftarrow e$. Notice that if a role name r is issuer-traces-none and subject-traces-none at the same time, nobody would have to store the credential $A.r \longleftarrow e$ (this is an ill-typed combination and will be ruled out in the next definition).

Let us go back to the two clauses in Example 9. We saw that if credential (1) was stored only by its subject and credential (2) was stored only by its issuer then any of the presented algorithms would be able to give a correct answer to the query "is D a member of $[\![A.r]\!]_{SP(\mathcal{C})}$?". In the light of Definition 5 this means that we have to avoid credentials of the form $A.r \longleftarrow B.r_1$, where $\mathit{itype}(r) = \textit{issuer-traces-none}$ and $\mathit{stype}(r) = \textit{subject-traces-none}$. In order to know which combinations are "good", we have the notion of *well-typed* credentials:

Definition 6 (Well-typed Credentials). *An RT_0 credential c is* well-typed *if no role name occurring in c has type $\{\textit{issuer-traces-none}, \textit{subject-traces-none}\}$ and:*

- *if $c = A.r \longleftarrow B.r_1$ then $\forall t \in \mathit{type}(r), \exists\, t_1 \in \mathit{type}(r_1)$ s.t. the corresponding entry in Table 1(a) is* OK*;*
- *if $c = A.r \longleftarrow A.r_1.r_2$ then $\forall t \in \mathit{type}(r), \exists\, t_1 \in \mathit{type}(r_1)$ and $\exists\, t_2 \in \mathit{type}(r_2)$ s.t. the corresponding entry in Table 1(b) is* OK*;*

– *if* $c = A.r \longleftarrow B_1.r_1 \cap B_2.r_2$ *then* $\forall t \in type(r), \exists\, t_1 \in type(r_1)$ *and* $\exists\, t_2 \in type(r_2)$ *s.t. the corresponding entry in Table 1(c) is* OK.

For example, take the credential $c : A.r \longleftarrow A.r_1.r_2$ and assume that $type(r) = type(r_1) = \{$*issuer-traces-def*, *subject-traces-all*$\}$ and that $type(r_2) = \{$*issuer-traces-none*, *subject-traces-all*$\}$. Then, we see that for both type values of r, *issuer-traces-def* and *subject-traces-all*, one can find a combination of type values for r_1 and r_2 such this combination appears as a valid type assignment in Table 1(b). For the issuer-side type value of r, *issuer-traces-def*, we have $itype(r_1) =$ *issuer-traces-def* and $stype(r_2) =$ *subject-traces-all*; for the subject-side type value of r, *subject-traces-all*, we have $stype(r_1) = stype(r_2) =$ *subject-traces-all*. On the other hand, if we had that $type(r) = type(r_1) = type(r_2) = \{$*issuer-traces-def*, *subject-traces-none*$\}$, then c would not be well typed as there is no valid entry for this type value assignment in Table 1(b). Note that simple member credentials (of the form $A.r \longleftarrow D$) are always well-typed.

The following theorem summarizes the results given in [25] and shows that using well-typed credentials guarantees that the algorithms presented in Sect. 3 give correct answers to queries even in presence of distributed credentials.

Theorem 1. *Let $\mathcal{C}$ be a set of well typed RT$_0$ credentials, and r be a role name.*

– *If $itype(r) =$ issuer-traces-all then for each entity A, the backward search algorithm correctly computes $[\![A.r]\!]_{SP(\mathcal{C})}$.*
– *If $stype(r) =$ subject-traces-all then for each entity D the forward search algorithm finds all the roles $A.r$ such that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$.*
– *For any given entity D, the bidirectional search algorithm can always correctly determine if $D \in [\![A.r]\!]_{SP(\mathcal{C})}$.*

Example 10. Consider again the policy of Example 9, if $type(r) = \{$*issuer-traces-all*, *subject-traces-none*$\}$ then, according to Table 1, for the credential (1) to be well-typed, the type of role name r_1 must also be $\{$*issuer-traces-all*, *subject-traces-none*$\}$. By Theorem 1, one can use the backward search algorithm to compute $[\![A.r]\!]_{SP(\mathcal{C})}$. On the other hand, if $type(r) = \{$*issuer-traces-none*, *subject-traces-all*$\}$ then, for the credential (1) to be well-typed, the type of r_1 must also be $\{$*issuer-traces-none*, *subject-traces-all*$\}$. For this type assignment, Theorem 1 says that, starting from D, the forward search algorithm will discover that D is a member of $B.r_1$ and $A.r$.

Finally, if $type(r) = \{$*issuer-traces-def*, *subject-traces-none*$\}$ and $type(r_1) = \{$*issuer-traces-none*, *subject-traces-all*$\}$ then one can check that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$ using the bidirectional search algorithm.

5 Other Members of the RT Family

As we have already mentioned, RT$_0$ is only one of the members of the RT family of TM languages. In this section we intend to give a flavour of these extensions and of the reasons why they have been introduced. We do so by presenting significative examples. For a full explanation of their syntax and semantics, we refer to [24] and [23].

5.1 RT_1

RT_1 extends RT_0 with parameterized roles. In RT_1 a role name consists of an RT_0 role name and zero or more parameters surrounded by parenthesis. A parameter can be a constant or a variable of one of five types: integer, closed enumeration, open enumeration, float, and date and type (see [24] for details).

Example 11. In CITA a project document can always be read and written by its author, no matter which policy applies to it. The remaining project members can read the document only if approved by the document author (identifiers starting with "?" are variables).

$$\begin{aligned} &\mathit{CITA.accessDoc(rw,?proj,?doc)} \longleftarrow \mathit{CITA.owner(?doc)} \cap \mathit{CITA.member(?proj)} \\ &\mathit{CITA.accessDoc(?access,?proj,?doc)} \longleftarrow \\ &\qquad\qquad \mathit{CITA.approved(?access,?doc)} \cap \mathit{CITA.member(?proj)} \\ &\mathit{CITA.approved(?access,?doc)} \longleftarrow \mathit{CITA.owner(?doc).approved(?access,?doc)} \end{aligned}$$

For each data type one can create a so called *static data set*, which can be used to constrain variables in credentials. Static in this context means that the values in the value set cannot depend on credentials but must be known at the time the value set is being specified.

Example 12. Charles restricts the access to his picture gallery to his friends that are over 18.

$$\mathit{Charles.accessPictures} \longleftarrow \mathit{Charles.friend(?Age:[18..100])}$$

In the example above, the possible values of the variable *?Age* are restricted to be in the range between 18 and 100.

For the linking inclusion credentials, a parameter can also be a special keyword *this*, which refers to a potential member of a linked role.

Example 13. CITA gives an annual salary increase to an employee if the employee's manager says that the performance of the employee is good.

$$\mathit{CITA.salaryIncrease} \longleftarrow \mathit{CITA.managerOf(this).goodPerformance}$$

5.2 RT_2

RT_2 extends RT_1 with *logical objects* that can be used to dynamically restrict possible values of the variables occurring in credentials. A logical object, or *o-set*, is similar to an RT_1 credential, but its member set is not restricted to that of entities. For instance, a company can define an o-set containing a selection of company's documents, running projects, and also any other valid RT_1 entity.

Example 14. The policy of CITA states that any document of a project in CUS is also a document of this project in CITA.

$$\mathit{CITA.document(?proj)} \longleftarrow \mathit{CUS.document(?proj)}$$

Now, CITA allows members of a project team to read documents of this project:

$$\mathit{CITA.accessDoc(read,?D{:}CITA.documents(?proj))} \longleftarrow \mathit{CITA.member(?proj)}$$

In the example above, *?D:CITA.documents(?proj)* shows the application of a dynamic value set. A dynamic value set is a generalization of the static value set of which example was given in Example 12. Similarly to the static value set, the dynamic value set can be used to constrain variables occurring in credentials. However, when the values in a static value set are fixed, the set of values a dynamic value set contains is given by the members of an o-set used as a constrain. In the example above, the set of values the variable *?D* can take is restricted to the members of the o-set *CITA.documents(?proj)*.

5.3 RT$^{\mathrm{T}}$

RT$^{\mathrm{T}}$ has been introduced to support threshold and separation of duty policies. Consider the following policy taken from [24]: "A says that an entity is a member of $A.r$ if one member of $A.r_1$ and two *different* members of $A.r_2$ all say so". This policy cannot be expressed in the RT dialects presented so far, and to express this in RT one needs to use the so-called *manifold roles*. Manifold roles extend the notion of roles by allowing role members to be *collections* of entities (rather than just principals). This is done in RT^T by defining the operators $\odot$ and $\otimes$. A credential of the form $A.r \longleftarrow B_1.r_1 \odot B_2.r_2$ says that $\{s_1 \cup s_2\}$ is a member of $A.r$ if s_1 is a member of $B_1.r_1$ and s_2 is a member of $B_2.r_2$. Notice that both s_1 and s_2 are (possibly singleton) sets of entities. A credential $A.r \longleftarrow B_1.r_1 \otimes B_2.r_2$ has a similar meaning, but it additionally requires that $s_1 \cap s_2 = \emptyset$. With these two additional sorts of credentials one can express the above statement as follows:

$$\begin{aligned} A.r &\longleftarrow A.r_4.r \\ A.r_4 &\longleftarrow A.r_1 \odot A.r_3 \\ A.r_3 &\longleftarrow A.r_2 \otimes A.r_2 \end{aligned}$$

Example 15. In CITA, a program must be verified by two *different* testers: one from CITA and one form CUS.

$$\begin{aligned} \mathit{CITA.verified} &\longleftarrow \mathit{CITA.testTeam.approved} \\ \mathit{CITA.testTeam} &\longleftarrow \mathit{CITA.tester} \otimes \mathit{CUS.tester} \end{aligned}$$

5.4 RT$^{\mathrm{D}}$

The RT framework also supports the so called *delegation of role activations*, which are useful when one needs to delegate authority temporarily to a process or an agent. RT$^{\mathrm{D}}$ provides a *delegation credential* for this reason. As delegation of role activation is a complex matter, here we only present the basic intuition of how it works. The simplest form of delegation credential is $D \xrightarrow{D \text{ as } A.r} B_0$, which means that D delegates to B_0 the right of acting in D's behalf "as member of $A.r$". We call "D as $A.r$" a *role activation*. In the delegation credential above, B_0 can also represent a request, rather than an entity. Consider for instance the following example:

Example 16. Frank is the general practitioner (GP) of Henk in the hospital of Enschede (Ziekenhuis Enschede – ZE). A general practitioner in ZE can access all medical records of his patients.

$$\mathit{ZE.gp(Henk)} \longleftarrow \mathit{Frank}$$
$$\mathit{ZE.accessMedRec(?Patient)} \longleftarrow \mathit{ZE.gp(?Patient)}$$

During his holiday in Poland, Henk had a serious accident and required immediate surgery in one of the hospitals in Warsaw (WH). Weronika, the operating doctor, needs to access Henk's medical records at ZE.

ZE and WH are members of the European Hospital Alliance (EHA). In case of necessity, a doctor from one of the associated hospitals can access the medical records of a patient of another hospital by activating her *emergency* role (this role is not active by default, and every activation is carefully logged in both the hospital and EHA logs).

$$\mathit{EHA.member} \longleftarrow \mathit{ZE}$$
$$\mathit{EHA.member} \longleftarrow \mathit{WH}$$
$$\mathit{ZE.accessMedRec(?Patient)} \longleftarrow \mathit{ZE.emergencyGroup.emergency(?Patient)}$$
$$\mathit{ZE.emergencyGroup} \longleftarrow \mathit{EHA.member}$$
$$\mathit{EHA.member} \longleftarrow \mathit{WH}$$
$$\mathit{WH.canActivateEmergency} \longleftarrow \mathit{WH.doctor}$$
$$\mathit{WH.doctor} \longleftarrow \mathit{Weronika}$$

Weronika can activate her role *WH.emergency(Henk)* and request Henk's medical records from ZA using the following delegation credential:

$$\mathit{Weronika} \xrightarrow{\mathit{Weronika\ as\ WH.emergency(Henk)}} \mathit{accessMedRec(ZE,Henk)}$$

Here notice that *accessMedRec(ZE,Henk)* is not an entity but represent an explicit request, which is then handled by a dummy entity in RT.

5.5 $\mathbf{RT}_\ominus$

The members of the RT family presented so far are monotonic: adding a credential to the system can only result in granting additional privileges. However, banishing negation from a TM language is not a realistic option. In fact, as stated by Li et al. [22]: "many security policies are non-monotonic, or more easily specified as non-monotonic ones". In [26], Czenko et al. argue that many access control decisions in complex distributed systems, like Virtual Communities (VC), are hard to model in a purely monotonic language. They propose $\mathrm{RT}_\ominus$, which adds to RT a restricted form of negation called *negation in context*.

$\mathrm{RT}_\ominus$ introduces a new operator $\ominus$ and the so called *exclusion* credential $A.r \longleftarrow B_1.r_1 \ominus B_2.r_2$ indicating that all members of $B.r_1$ which are *not* members of $B_2.r_2$ are members of $A.r$.

Example 17. Consider the policy of Example 5. In this policy Charles's role *friends* is defined to be a transitive closure of the set of his direct friends. Now, if for some reason Charles would like to *exclude* some entities from this set, he needs to use the following exclusion credential:

$$\textit{Charles.accessPictures} \longleftarrow \textit{Charles.friend} \ominus \textit{Charles.blackList}$$

Now, an entity is a member of Charles's *accessPicture* role if she is a member of Charles's role *friend* and she is *not* on the Charles's black list. Assume that we have:

$$\textit{Charles.blackList} \longleftarrow \textit{Sandro}$$

Then the semantics of the role *Charles.accessPictures* is:

$$[\![\textit{Charles.accessPictures}]\!]_{SP(\mathcal{C})} = \{\textit{Alice, Bob, Jeffrey, Johan}\}.$$

5.6 Summary

The table below summarizes the key features of all the members of the RT framework.

The RT family member	Key extensions
RT_1	parameterized roles
RT_2	logical objects
RT^{T}	manifold roles and role-product operators, which can express threshold and separation of duty policies
RT^{D}	delegation of role activation, which allows for selective use of credentials
$\mathrm{RT}_{\ominus}$	restricted form of negation

RT^{D} and RT^{T} can be used, together or separately, in combination with either RT_0, RT_1, or RT_2. The resulting combinations are written $\mathrm{RT}_i^{\mathrm{D}}$, $\mathrm{RT}_i^{\mathrm{T}}$, and $\mathrm{RT}_i^{\mathrm{DT}}$ for $i = 0, 1, 2$.

6 Related Work

6.1 Trust Management Systems

PolicyMaker and KeyNote. The notion of trust management was introduced by Blaze et al. [10], as a problem in network security for which the authors proposed an approach based on a small collection of general principals: unified mechanism, flexibility (expressiveness), locality of control (autonomy of system participants), and separation of policy from mechanism. PolicyMaker, also designed and developed by Blaze et al. [9,10], was the first trust management prototype system that "facilitates the development of security features in a wide range of network services." [10]

Unlike RT, PolicyMaker places very few restrictions on the specification of authorizations and delegations. Policies and credentials are fully programmable, and can be arbitrary executable programs, limited only by being strongly "sandboxed." The advantage is that the PolicyMaker approach enables application developers tremendous

flexibility to define authorizations and delegations. However, its compliance checking (evaluation) is in general undecidable: no algorithm can, for each possible request, decide whether the request is authorized. There are several variants of PolicyMaker's proof of compliance problem that are proven to be decidable, but NP-hard: globally bounded proof of compliance (GBPOC), locally bounded proof of compliance (LBPOC), and monotonic proof of compliance(MPOC). A polynomial time bound can be achieved for compliance checking by combining the restrictions used in LBPOC with the requirement that assertions be monotonic. However, the constant parameters that limit computational effort expended by a legal proof of compliance are imposed arbitrarily without apparently natural justification.

KeyNote [8] is a direct descendant of PolicyMaker. KeyNote's assertions are written in a concise and human readable assertion language. Evaluation is based on expression evaluation, rather than on the execution of arbitrary programs, and is specified by an informal, implementation-independent semantics that defines authorization decisions based on requested actions. Action requests are represented by a collection of variable bindings, and credentials can contain constraints on these variables that can be used to restrict the actions for which credential owners are authorized.

Credentials, in both PolicyMaker and KeyNote, bind public keys (of the credential subjects) to direct authorizations of security-critical actions. Therefore, similarly to capability-based system, KeyNote's authorization decision procedure is quite straightforward, without necessarily resolving the name or identity of the requester. However, capability-based systems are not as scalable as attribute-based systems. In capability-based systems, managing the delegation of access rights, for instance, to all students at a given university requires issuing a credential to each student for each resource to which they have access (library, cafeteria, gym, etc.). In attribute-based systems, such as RT, by utilizing credentials that characterize their owners as being students, the same student ID credential can be used to authorize a wide range of actions.

SPKI/SDSI. SPKI/SDSI [12] merged the SDSI [29] and the SPKI [14] efforts together to achieve an expressive and powerful trust management system. SDSI (pronounced “sudsy"), short for “a Simple Distributed Security Infrastructure," was proposed as a new public-key infrastructure by Lampson and Rivest. Concurrently, Carl Ellison et al. developed SPKI (pronounced “spooky"), which was an abbreviation for “Simple Public Key Infrastructure."

SDSI's main contribution is its design of linked local names, which solves the problem of determining globally unique names. In SDSI, the owner of each public key can define names local to a name space that is identified by that key. For example, “K_{Alice} friends" represents a SDSI name, where K_{Alice} is a key identifying its name space and “friends" is a name defined locally in that name space by K_{Alice}. SDSI names that start with different keys are different names, so there is no danger that local names in different name spaces will interfere with one another. In this way, global uniqueness of names is achieved without synchronizing and coordinating naming authorities. The way in which RT's roles are defined locally, but can be referenced non-locally, is inherited from SDSI's design of local name spaces.

While SDSI is responsible for binding names to public keys, SPKI is responsible for making authorizations. SPKI's authorization scheme can be regarded as being

orthogonal to SDSI's naming scheme. Originally in SPKI, the certificate subject is represented by its public key. However, in SPKI/SDSI, the subject can be represented by its SDSI name. SDSI names provide a method to define *groups* of authorized principals, which simplifies the delegation procedure.

For example, if Bob wants to grant an authorization to Alice's friends, Bob can simply use SDSI's group name "K_{Alice} friends". By contrast, using KeyNote, Bob would have to enumerate the public keys of every friend of Alice's in the "Licensee" field of the assertion. The flexibility obtained by using SDSI names is useful in a decentralized system. On one hand, Bob does not need to have a list of Alice's friends when he is writing the authorization policies. On the other hand, any changes on Alice's friends list will be immediately reflected in the semantics of Bob's authorization policies.

SPKI/SDSI's evaluator uses a bottom-up algorithm to compute a closure set containing all certificates that can be derived from the given set of certificates. A request can be authorized if it can be found in the closure set. This algorithm is proven to be polynomial [12]. However, the evaluation process must be repeated whenever any certificate has been added or revoked, or has expired, so it is not suitable for use with a large and frequently changing credential pool.

Cassandra. Cassandra [6,7] is a role-based trust management system, which was designed with the goal of supporting the access control policies for a national electronic health record (EHR) system.

Like RT^C, Cassandra represents policy statements in Datalog clauses with constraints. Six special predicates are predefined in Cassandra. Firstly, *canActivate*(e, r) expresses that entity e can activate role r and, as such, that e is a member of r. Secondly, *hasActivated*(e, r) indicates that entity e has activated role r. The distinction between the predicates *canActivate* and *hasActivated* corresponds to the distinction between the role membership and the session activation in traditional RBAC [2]. Thirdly, *canDeactivate*(e_1, e_2, r) holds if entity e_1 has the power to deactivate e_2's activation of role r. Fourthly, *isDeactivated*(e, r) becomes true if entity e's role r is deactivated. Therefore, unlike RT that can only support role membership, Cassandra can also express role activations and deactivations. If a role is activated by a principal, a new fact (i.e., an atomic formula) representing this activation, and using predicate *hasActivated*, is put into the policy; similarly, deactivation of roles causes facts with predicate *hasActivated* to be removed from the policy. Fifthly, *permits*(e, a) says that the entity e is permitted to perform action a. This differs from the standard notion of role-permission assignment in two ways. On one hand, the parameter e allows constraints to refer directly to the subject of the activation. On the other hand, *permits* has no parameter for a role associated with the action, thus allowing more flexible permission specifications, e.g., a permission that is conditioned on the activation or (or perhaps merely membership in) more than one single role. Finally, *canReqCred*$(e_1, e_2.p(\overrightarrow{e}))$ says that the entity e_1 is allowed to request credentials issued by the entity e_2 and asserting the predicate $p(\overrightarrow{e})$. Besides these six special predicates, application developers can also define their own customized predicates.

TPL. TPL (Trust Policy Language) [17], designed at IBM Haifa Research Lab, was proposed specifically for trust establishment between e-strangers. TPL is based on

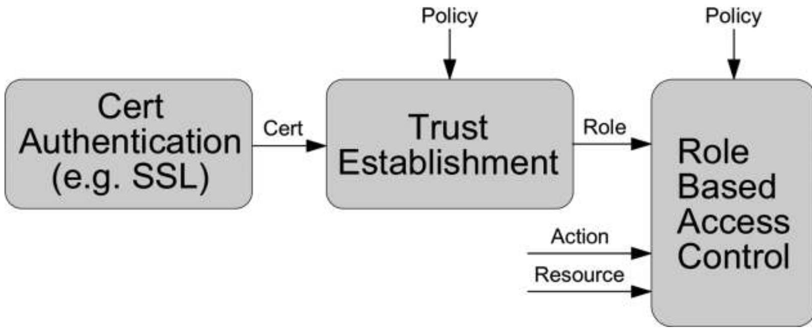


Fig. 4. TCP

RBAC [2] and extends it by being able to map strangers to roles. Unlike RT and Cassandra, TPL's efforts are claimed to be put only into mapping users to roles, but not into mapping roles to privileges, which simplifies the design. Figure 4 shows the relations between TPL and RBAC that TPL works in the Trust Establishment module and transfers the resolved role names to RBAC module.

TPL uses XML for application developers to write security rules, which will be translated in TPL to a standard logic programming language, e.g. Prolog. Unlike RT, which is monotonic, TPL is non-monotonic, since it includes negative rules. A negative rule indicates that learning a new piece of knowledge (e.g., a credential) will reduce the requester's privileges. For example, a negative rule represented in Prolog statement can be "$\textit{group}(X, \text{Discount}) :- \backslash + \textit{group}(X, \text{Felon})$," in which "$\backslash +$" represents the *negation of failure*. It means that if the credential of being a felon is failed to be derived, then the requester is allowed to have the discount. However, the completeness and soundness of TPL are not specified in the original work. The example below [17] shows a rule written in XML and its Prolog translation.

XML:
<GROUP NAME="Hospitals">
<RULE>
<INCLUSION **ID**="reco" TYPE="Recommendation" FROM="self"/>
</RULE>
</GROUP>

Prolog:
$group(?X, \textit{Hospitals}) :- cert(?Y, ?X, \textit{Recommendation}, \textit{_RecFields}),$
$group(?Y, \textit{self}).$

PCA. PCA (Proof Carrying Authorization) [3,5,4] was mainly designed for the access control on server's web page resources. Figure 5 shows the components of PCA system working in a web browsing environment. $HTTP\ proxy$ is used to make the whole process of accessing a web page transparent to the web browser. The web browser only knows the final result: either the requested web page or a denial message is displayed.

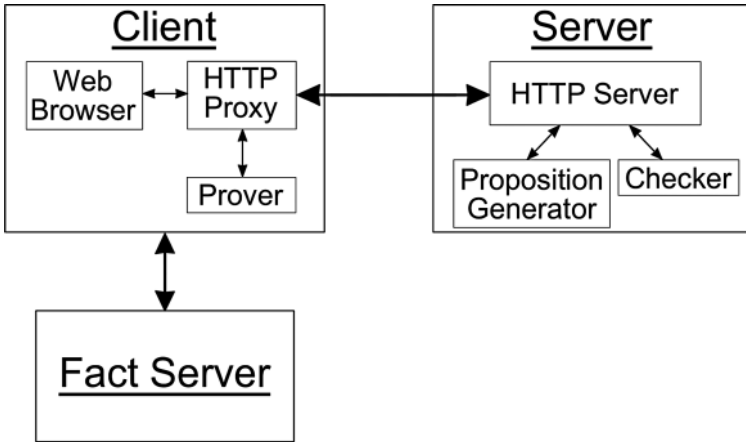


Fig. 5. PCA system

The proxy is designed to be portable and easily integrated into the client system without changing anything inside the original web browser. Therefore, it facilitates the client who is not knowledgeable in collecting relevant credentials and negotiating with the resource owner.

PCA uses higher-order logic to specify policies and credentials, so that it can be very expressive. However, its evaluation is thus undecidable. In their design, undecidability is resolved in two phases. Firstly, in order to reduce the computation burden on the server's PCA evaluator, it is required that the requesting client constructs the proof. The server's evaluator only needs to check the proof, which is not only decidable, but can be done quite efficiently. Secondly, on the client side, the proxy is responsible for navigating and retrieving credentials, computing proofs and communicating with the server. In order to avoid undecidable computation at the client side, the client proxy does not use the full logic, but use an application-specific limited logic, which should be tractable.

QCM. QCM [16], short for "Query Certificate Manager", was designed at the University of Pennsylvania as a part of the SwitchWare project on active networks to support secure maintenance of distributed data sets. For example, QCM can be used to support decentralized administration of distributed repositories housing public key certificates that map names to public keys. For the purposes of access control, QCM provides security support for ACL's query and retrieval.

QCM's policy is specified in relational calculus. One of the main contributions of QCM is its design of a policy directed certificate retrieval mechanism [16], which enables the TM evaluator automatically to detect and identify missing but needed certificates, and to retrieve them from remote certificate repositories. It uses query decomposition and optimization techniques, and its novel solutions are discussed in terms of network security, such as private key protection methods. However, unlike RT credentials, which can be stored with their either their subjects or their issuers, and can then

Fig. 6. QCM Engine

be located and retrieved as needed during authorization evaluation, credentials in QCM must be stored with their issuers. Figure 6 [16] shows the stages by which queries and credentials are processed in the evaluator and how the evaluation process interleaves and cooperates with the credential retrieval process.

6.2 Trust Management and Reputation Systems

Since the term Trust Management was first introduced by Blaze et al. in [10], TM became an important and popular research area. However, in many cases the work having TM in the title has often very little in common with TM as understood by its originators. Most of these cases come from the field of *Reputation Systems* [28,34,33,30,21] – also referred in the literature as *Reputation Based Trust Management*. In this paper we do not deal with reputation systems, however, as reputation systems are undoubtedly related to TM, below we provide some background information and we mark the most evident differences with TM.

Reputation systems is now a well researched area [35]. The interest in reputation systems comes from e.g. expert and auction systems [28], like*AllExperts* (http://www.allexperts.com), where everyone can ask an expert volunteer a question from the selected area. The user can then rate the expert so that other users be informed on the quality of advice given by different experts. An example of an auction system is *eBay* (http://www.ebay.com). In eBay, every user is welcome to leave a positive, negative or neutral feedback after each transaction. Sellers and buyers in eBay can rate each

others and by this they can discourage (or encourage) prospective users to enter into business with another eBay user.

It has been observed that reputation is an important factor which naturally supports the process of building trust among people [18,28]. The role of a reputation system is then to collect, distribute, and aggregate feedbacks (reputations) concerning participants' past behavior [28]. The past behavior is usually expressed using a so called *trust metric*, which describes the agent's trust in another agent - most often within some well defined context [1]. In defining trust and reputation, authors often refer to social sciences [1,27] or economy and politics [28,13]. In most of the formal approaches to reputation based trust management there is a clear distinction between a so called *direct* and *recommendation* trust [35,20,1,34].

It is clear that the areas that both Trust Management and Reputation Systems cover overlap. There are, however, important differences. Most reputation systems are numeric [11], and do not incorporate language facilities. Reputation systems are also in general highly dynamic and deal mostly with the trust metric definition or recommendation exchange protocols. Reputation systems answer the question how to build trust values from the local history and the information provided by other peers. Most importantly, the trust gained in reputation systems is rather fuzzy in nature as it depends on an often obscure algorithm and on sometimes highly subjective feedback. In Trust Management, on the other hand, trust is obtained as a result of a formal evaluation of a set of credentials with respect to the user policy. Each user is also allowed to have different policy, which is usually not allowed in the existing reputation systems.

References

1. Abdul-Rahman, A., Hailes, S.: Supporting Trust in Virtual Communities. In: Proc. 33rd Hawaii International Conference on System Sciences, vol. 6, p. 6007. IEEE Computer Society Press, Los Alamitos (2000)
2. ANSI: American National Standard for Information Technology – Role Based Access Control. ANSI INCITS 359-2004 (February 2004)
3. Appel, A.W., Felten, E.W.: Proof-Carrying Authentication. In: CCS '99: Proc. 6th ACM Conference on Computer and Communications Security, pp. 52–62. ACM Press, New York (1999)
4. Bauer, L., Schneider, M.A., Felten, E.W.: A General and Flexible Access-Control System for the Web. In: Proc. 11th USENIX Security Symposium, USENIX Association, pp. 93–108 (2002)
5. Bauer, L.: Access Control for the Web via Proof-Carrying Authorization. PhD thesis, Adviser-Andrew W. Appel. (2003)
6. Becker, M.Y., Sewell, P.: Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In: Proc. 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), pp. 159–168. IEEE Computer Society Press, Los Alamitos (2004)
7. Becker, M.Y., Sewell, P.: Cassandra: Flexible Trust Management, Applied to Electronic Health Records. In: CSFW, pp. 139–154. IEEE Computer Society Press, Los Alamitos (2004)
8. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The KeyNote Trust-Management System, Version 2. IETF RFC 2704 (1999)

9. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The Role of Trust Management in Distributed Systems Security. In: Vitek, J., Jensen, C. (eds.) Secure Internet Programming. LNCS, vol. 1603, pp. 185–210. Springer, Heidelberg (1999)
10. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized Trust Management. In: Proc. 17th IEEE Symposium on Security and Privacy, pp. 164–173. IEEE Computer Society Press, Los Alamitos (1996)
11. Bonatti, P., Duma, C., Olemdilla, D., Shahmehri, N.: An Integration of Reputation-based and Policy-based Trust Management. In: Proc. Semantic Web and Policy Workshop (2005)
12. Clarke, D., Elien, J.E., Ellison, C., Fredette, M., Morcos, A., Rivest, R.L.: Certificate Chain Discovery in SPKI/SDSI. Journal of Computer Security 9(4), 285–322 (2001)
13. Dellarocas, C.: Analyzing the Economic Efficiency of eBay-like Online Reputation Reporting Mechanisms. In: Proc. 3rd ACM conference on Electronic Commerce, pp. 171–179. ACM Press, New York (2001)
14. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., Ylonen, T.: SPKI Certificate Theory. IETF RFC 2693 (September 1999)
15. Etalle, S., Winsborough, W.H.: A Posteriori Compliance Control. In: Proc. 12th ACM Symposium on Access Control Models and Technologies, ACM Press, New York (2007)
16. Gunter, C., Jim, T.: Policy-directed Certificate Retrieval. Software: Practice & Experience 30(15), 1609–1640 (2000)
17. Herzberg, A., Mass, Y., Michaeli, J., Ravid, Y., Naor, D.: Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. In: Proc. IEEE Symposium on Security and Privacy, pp. 2–14. IEEE Computer Society Press, Los Alamitos (2000)
18. Jarvenpaa, S.L., Tractinsky, N., Vitale, M.: Consumer Trust in an Internet Store. Inf. Tech. and Management 1(1-2), 45–71 (2000)
19. Jim, T.: SD3: A Trust Management System with Certified Evaluation. In: Proc. IEEE Symposium on Security and Privacy, pp. 106–115. IEEE Computer Society Press, Los Alamitos (2001)
20. Jøsang, A.: The Right Type of Trust for Distributed Systems. In: NSPW '96: Proc. Workshop on New Security Paradigms, pp. 119–131. ACM Press, New York (1996)
21. Kamvar, S.D., Schlosser, M.T., Garcia-Molina, H.: The Eigentrust Algorithm for Reputation Management in P2P Networks. In: Proc. 12th International Conference on World Wide Web, pp. 640–651. ACM Press, New York (2003)
22. Li, N., Feigenbaum, J., Grosof, B.N.: A Logic-based Knowledge Representation for Authorization with Delegation (Extended Abstract). In: Proc. 1999 IEEE Computer Security Foundations Workshop, pp. 162–174. IEEE Computer Society Press, Los Alamitos (1999)
23. Li, N., Mitchell, J.: RT: A Role-based Trust-management Framework. In: Proc. 3rd DARPA Information Survivability Conference and Exposition (DISCEX III), pp. 201–212. IEEE Computer Society Press, Los Alamitos (2003)
24. Li, N., Mitchell, J., Winsborough, W.: Design of a Role-based Trust-management Framework. In: Proc. IEEE Symposium on Security and Privacy, pp. 114–130. IEEE Computer Society Press, Los Alamitos (2002)
25. Li, N., Winsborough, W., Mitchell, J.: Distributed Credential Chain Discovery in Trust Management. Journal of Computer Security 11(1), 35–86 (2003)
26. Czenko, M., Tran, H., Doumen, J., Etalle, S., Hartel, P., den Hartog, J.: Nonmonotonic Trust Management for P2P Applications. In: Proc. 1st International Workshop on Security and Trust Management, pp. 101–116. Elsevier, Amsterdam (2005)
27. Mui, L., Mohtashemi, M., Halberstadt, A.: A Computational Model of Trust and Reputation for E-businesses. Hicss 07, 188 (2002)
28. Resnick, P., Kuwabara, K., Zeckhauser, R., Friedman, E.: Reputation systems. Commun. ACM 43(12), 45–48 (2000)

29. Rivest, R., Lampson, B.: SDSI – A Simple Distributed Security Infrastructure (October 1996), Available at
http://theory.lcs.mit.edu/~rivest/sdsi11.html
30. Shmatikov, V., Talcott, C.L.: Reputation-based Trust Management. Journal of Computer Security 13(1), 167–190 (2005)
31. Weeks, S.: Understanding Trust Management Systems. In: Proc. IEEE Symposium on Security and Privacy, pp. 94–105. IEEE Computer Society Press, Los Alamitos (2001)
32. Winsborough, W.H., Li, N.: Towards Practical Automated Trust Negotiation. In: POLICY, pp. 92–103. IEEE Computer Society Press, Los Alamitos (2002)
33. Xiong, L., Liu, L.: A Reputation-based Trust Model for Peer-to-Peer eCommerce Communities. In: ACM Conference on Electronic Commerce, pp. 228–229. ACM, New York (2003)
34. Xiong, L., Liu, L.: PeerTrust: Supporting Reputation-Based Trust for Peer-to-Peer Electronic Communities. IEEE Trans. Knowl. Data Eng. 16(7), 843–857 (2004)
35. Yahalom, R., Klein, B., Beth, T.: Trust Relationships in Secure Systems – A Distributed Authentication Perspective. In: RSP: IEEE Computer Society Symposium on Research in Security and Privacy, IEEE Computer Society, Los Alamitos (1993)