Design of a Role-based Trust-management Framework

Ninghui Li John C. Mitchell
Department of Computer Science
Stanford University
Gates 4B
Stanford, CA 94305-9045
{ninghui.li, mitchell}@cs.stanford.edu

William H. Winsborough
NAI Labs
Network Associates, Inc.
3060 Washington Road
Glenwood, MD 21738
william_winsborough@nai.com

Abstract

We introduce the RT framework, a family of Role-based Trust-management languages for representing policies and credentials in distributed authorization. RT combines the strengths of role-based access control and trust-management systems and is especially suitable for attribute-based access control. Using a few simple credential forms, RT provides localized authority over roles, delegation in role definition, linked roles, and parameterized roles. RT also introduces manifold roles, which can be used to express threshold and separation-of-duty policies, and delegation of role activations. We formally define the semantics of credentials in the RT framework by presenting a translation from credentials to Datalog rules. This translation also shows that this semantics is algorithmically tractable.

1 Introduction

We introduce the RT framework, a family of Role-based Trust-management languages for representing policies and credentials in distributed authorization. Development of the RT framework is part of an on-going effort to address security problems that arise when independent organizations enter into coalitions whose membership and very existence change rapidly. A coalition may be formed by several autonomous organizations wishing to share resources. While sharing resources, each organization retains ultimate authority over the resources it controlled prior to entering the coalition. We call such systems *decentralized collaborative systems*, since they have no single central authority.

Access control in decentralized collaborative systems presents difficult problems, particularly when resources and the subjects requesting them belong to different security domains controlled by different authorities. Traditional access control mechanisms make authorization decisions based on the identity of the resource requester. Unfortunately, when the resource owner and the requester are unknown to one another, access control based on identity may be ineffective. In [3], Blaze, Feigenbaum, and Lacy coined the term "trust management" to group together some principles dealing with decentralized authorization.

Some trust management (TM) systems, such as KeyNote [2] and SPKI/SDSI [8, 10], use credentials to delegate permissions. Each credential delegates certain permissions from its issuer to its subject. A chain of one or more credentials acts as a capability, granting certain permissions to the subject of the last credential in the chain. However, even such capability-style systems do not address the distributed nature of authority in a decentralized environment.

Consider an simple example: A book store wants to give 15% discount to students of a nearby university. Ideally, the book store would express this policy in one statement, and a student could get the discount by showing her student ID, issued by the university. However, one cannot follow this simple approach in capability-style TM systems. For instance, in KeyNote or SPKI 1.0[1], one cannot express the statement that anyone who is a student is entitled to a discount. Instead, there are two alternative approaches, neither of which is satisfactory. One is to have the book store delegate the discount permission to the university's key. Then the university's key explicitly delegates this permission to each student's key. This places too heavy an administrative burden on the university, since there could be many businesses giving discounts to students, each of which requires a separate delegation by the university to each student. In the second approach, the university would create a new key pair representing the group of all students. Each student ID would then be a complete delegation from this group key to the student's public key. This would allow the bookstore to establish the student discount policy by is-

[1]We use SPKI 1.0 to denote the part of SPKI/SDSI 2.0 [8, 10] originally from SPKI, *i.e.*, 5-tuples, and SDSI 1.0 to denote the part of SDSI originally from SDSI, *i.e.*, name certificates (or 4-tuples as called in [10]).

suing a credential granting the discount to the group key. However, this approach requires the university to manage a separate public/private key pair for each meaningful group. It also requires group public keys to be distributed to entities like book stores. Furthermore, the bookstore needs to know which key corresponds to the group of students; this would require another public-key infrastructure. Moreover, using one key pair to represent each group provides very limited expressive power. A student ID takes the form of a complete delegation from the group key and cannot contain any application-independent attribute information about the student, such as department, year, *etc.*, which are often useful in deriving other attributes or making access control decisions.

To simplify authorization in collaborative environments, we need a system in which access control decisions are based on authenticated attributes of the subjects, and attribute authority is decentralized. We call such systems attribute-based access control (ABAC) systems. We argue that an expressive ABAC system should be able to express the following:

1. Decentralized attributes: an entity asserts that another entity has a certain attribute.
2. Delegation of attribute authority: an entity delegates the authority over an attribute to another entity, *i.e.*, the entity trusts another entity's judgement on the attribute.
3. Inference of attributes: an entity uses one attribute to make inferences about another attribute.
4. Attribute fields. It is often useful to have attribute credentials carry field values, such as age and credit limit. It is also useful to infer additional attributes based on these field values and to delegate attribute authority to a certain entity only for certain specific field values, *e.g.*, only when spending level is below a certain limit.
5. Attribute-based delegation of attribute authority. A key to ABAC's scalability is the ability to delegate to strangers whose trustworthiness is determined based on their own certified attributes. For example, one may delegate the authority on (identifying) students to entities that are certified universities, and delegate the authority on universities to an accrediting board. By doing so, one avoids having to know all the universities.

Using KeyNote, SPKI 1.0, or X.509 attribute certificates [11], one cannot express inference of attributes or attribute-based delegation. SDSI 1.0 or even SPKI/SDSI 2.0 do not support attribute fields. Neither TPL [15] nor the language in [4] supports delegation of authority over arbitrary attributes. Although one can use Delegation Logic (DL) [17, 19] to express all of the above, it is not very convenient. Through a basic attribute credential, a designated issuer should be able to express the judgement that a subject has a certain attribute. A basic certificate in DL has only an issuer and a statement. Although one can encode the subject and attribute together in a statement, DL lacks the explicit subject abstraction, which we desire for the following reasons. The explicit abstraction allows clear, concise representation of attribute-based delegation, *e.g.*, in the form of linked local names in SDSI. The subject abstraction also enables distributed storage and discovery of credentials, as shown in [20]. It also enables us to view attributes similarly to roles in role-based access control (RBAC) [23], and to use concepts similar to role activations to enable entities to make selective use of those roles. Another TM system SD3 [16] can be viewed as Delegation Logic without delegation constructs; it does not have the subject abstraction either.

RT is our proposal for meeting the requirements of ABAC systems. RT uses the notion of roles to represent attributes. A *role* in RT defines a set of entities who are members of this role. A role can be viewed as an *attribute*: An entity is a member of a role if and only if it has the attribute identified by the role. This notion of roles also captures the notions of groups in many systems.

RT combines the strengths of RBAC and trust-management (TM) systems. From RBAC, it borrows the notions of role, interposed in the assignment of permissions to users to aid organizing those assignments, and of sessions and selective role activations. From TM, RT borrows principles of managing distributed authority through the use of credentials, as well as some clear notation denoting relationships between those authorities, *e.g.*, localized name spaces and linked local names from SDSI. From DL, it borrows the logic-programming-based approach to TM. In addition, RT has policy concepts such as intersections of roles, role-product operators, manifold roles, and delegation of role activations. These concepts can express policies that are not possible to express in existing systems; they can also express some other policies in more succinct or intuitive ways.

The most basic part of RT, RT_0, was presented in [20], together with algorithms that search for chains of RT_0 credentials, and a type system about credential storage that ensures chains can be found among credentials whose storage is distributed. RT_0 meets four of the five requirements listed above; it doesn't supports attribute fields. In this paper, we present four additional components of the RT framework: RT_1, RT_2, RT^T, and RT^D. RT_1 adds to RT_0 parameterized roles, which can express attribute fields. RT_2 adds to RT_1 logical objects, which can group logically related objects together so that permissions about them can be assigned together. RT^T provides manifold roles and role-product operators, which can express threshold and separation-of-duty policies. RT^D provides delegation of role activations, which can express selective use of capacities and delegation of these capacities.

Our goal in designing RT is an expressive yet clean system with an intuitive, formally defined, and tractable semantics. We formally define the semantics of RT credentials by presenting a translation from credentials to negation-free, safe Datalog rules. This guarantees that the semantics is precise, monotonic, and algorithmically tractable. In decentralized collaborative systems, there must be agreement on the meaning of credentials. Because Datalog is a subset of first-order predicate calculus, it provides a clear semantic point of reference. Monotonicity of the semantics is important, especially in distributed environments, since to deal with non-monotonicity, one typically needs complete information, which is very hard to obtain in distributed environments. A Datalog rule is safe if all variables in its head also appear in the body. This guarantees that the set of conclusions it generates is finite and bounded. The requirement that each RT credential can be translated to a negation-free, safe Datalog rule is the main constraint on expressivity guiding the design of RT features presented here.

The rest of this paper is organized as follows. In section 2, we give an overview of the main features and components of RT. In sections 3, 4, 5, and 6, we introduce four components: RT_1, RT_2, RT^T, and RT^D, respectively. We then discuss the current status of RT, implementation issues, future and related work in section 7, and conclude in section 8.

2 An Overview to the RT framework

In this section, we introduce core concepts in the RT framework and summarize the different components in RT.

2.1 Entities and roles

An *entity* in RT is a uniquely identified individual or process. Entities are also called principals in the literature. They can issue credentials and make requests. RT assumes that one can determine which entity issued a particular credential or a request. Public/private key pairs clearly make this possible. In some environments, an entity could also be, say, a secret key or a user account. In this paper, we use A, B, and D, sometimes with subscripts, to denote entities. A *role* in RT defines a set of entities who are members of this role.

In role-based access control (RBAC) [23], there are users, roles, and permissions. Roles in RBAC form a middle layer between users and permissions, thus simplifying the management of the many-to-many relationships between users and permissions. Permissions are associated with roles, and users are granted membership in appropriate roles, thereby acquiring the roles' permissions. More advanced RBAC models include role hierarchies. A role hierarchy extends the role layer to have multiple layers, thereby further reducing the number of relationships that must be managed. Role hierarchies are partial orders. If one role, r_1, *dominates* another, r_2, (written $r_1 \succeq r_2$), then r_1 has every permission that r_2 has.

Entities in RT correspond to users in RBAC. Roles in RT can represent both roles and permissions from RBAC. In RT, we view user-role assignments and role-permission assignments also as domination relationships. Assigning a user u to a role r can be represented as $u \succeq r$. And assigning a permission p to a role r can be represented as $r \succeq p$. In this way, user-role assignments, the role hierarchy, and role-permission assignments together define a uniform partial order over users, roles, and permissions. In addition to its implications for permissions, $r_1 \succeq r_2$ can equivalently be read as implying that any user who is a member of r_1 is also automatically a member of r_2. We say that r_2 *contains* (all the user members of) r_1. The contains ordering is the inverse of the dominates ordering. A partial order can be defined using either. RT uses the contains ordering, because it is entity centric and corresponds better to the attribute interpretation of roles.

2.2 Localized authority for roles

RBAC was developed for access control in a single organization. Some authors consider it an essential characteristic of RBAC that the control of role membership and role permissions be relatively centralized in a few users [23]. This centralized control feature does not work well in distributed collaborative systems. To handle the decentralized nature of distributed collaborative systems, RT borrows from existing trust-management systems, since the focus of these systems is decentralized control. In particular, we borrow from SDSI the concepts of localized name spaces. In SDSI, a local name is formed by an entity followed by a name identifier. Each entity has its own localized name space and is responsible for defining local names in its own name space.

In RT, a role is denoted by an entity followed by a role name, separated by a dot. We use R, often with subscripts, to denote role names. A role $A.R$ can be read as A's R role. Only A has the authority to define the members of the role $A.R$, and A does so by issuing role-definition credentials. Each credential defines one role to contain either an entity, another role, or certain other expressions that evaluate to a set of entities. A role may be defined by multiple credentials. Their effect is union.

An entity A can define $A.R$ to contain $A.R_1$, another role defined by A. Such a credential reads $A.R \longleftarrow A.R_1$; it means that A defines that R_1 dominates R. At the same time, a credential $A.R \longleftarrow B.R$ is a delegation from A to B of authority over R. This can be used to decentralize the user-role assignment. A credential of the form $A.R \longleftarrow B.R_1$ can be used to define role-mapping across multiple organizations when they collaborate; it also represents a delegation from A to B.

Using a linked role in a credential enables the issuer to delegate to each member of a role. The credential $A.R \longleftarrow A.R_1.R_2$ states that: $A.R$ contains any $B.R_2$ if $A.R_1$ contains B. The set of contains relationships implied by this credential is dynamic in that it depends upon other credentials, some present at the time when this credential is issued and any others issued later. RT also allows the use of intersection while defining roles. A credential $A.R \longleftarrow B_1.R_1 \cap B_2.R_2$ states that: $A.R$ contains any role or entity that are contained by both $B_1.R_1$ and $B_2.R_2$. The set of contains relationships implied by this credential is also dynamic. Note that such role intersections do not exist in RBAC. The following example from [20] illustrates the use of RT_0 credentials.

Example 1 A fictitious Web publishing service, EPub, offers a discount to anyone who is both a preferred customer and a student. EPub delegates the authority over the identification of preferred customers to its parent organization, EOrg. EOrg issues a credential stating that IEEE members are preferred customers. EPub delegates the authority over the identification of students to entities that are accredited universities. To identify such universities, EPub accepts accrediting credentials issued by the fictitious Accrediting Board for Universities (ABU). The following credentials prove that Alice is eligible for the discount:

$$\left\{ \begin{array}{c} \mathtt{EPub.disct} \longleftarrow \mathtt{EPub.preferred} \cap \mathtt{EPub.student} \\ \mathtt{EPub.preferred} \longleftarrow \mathtt{EOrg.preferred} \\ \mathtt{EOrg.preferred} \longleftarrow \mathtt{IEEE.member} \\ \mathtt{EPub.student} \longleftarrow \mathtt{EPub.university.stuID} \\ \mathtt{EPub.university} \longleftarrow \mathtt{ABU.accredited} \\ \mathtt{ABU.accredited} \longleftarrow \mathtt{StateU} \\ \mathtt{StateU.stuID} \longleftarrow \mathtt{Alice},\ \mathtt{IEEE.member} \longleftarrow \mathtt{Alice} \end{array} \right\}$$

2.3 Parameterized roles

In RBAC, a role name is an atomic string. It has been noted in the literature that this is sometimes too limited [12, 21]. An organization may contain large numbers of roles with few differences between them. For example, each project has a project leader role, and the rights of project leaders over documents of their projects is often the same. It is desirable to facilitate the reuse of these role permission relationships. To address this, the notion of parameterized roles was introduced in [12, 21] (it was called role templates in [12]).

RT_0 only allows atomic strings as role names. RT_1 extends RT_0 to allow parameterized roles. In RT_1, a *role name* is constructed by applying a *role identifier* to a tuple of data terms. In this paper, we use r, often with subscripts, to denote role identifiers, and use h, s, and t with subscripts to denote data terms. RT_1 will be introduced in section 3.

Parameterized roles can represent relationships between entities. For example, if a company Alpha allows the manager of an employee to evaluate an employee (or maybe approve a purchase request submitted by the employee); we can use Alpha.managerOf(employee) to name the manager of an employee. Similarly, if a hospital Beta allows the physician of a patient to access the documents of a patient, we can use Beta.physicianOf(patient) to name the physician of a patient. Parameterized roles can also represent attributes that have fields. For example, a diploma typically contain school, degree, year, *etc.* An IEEE membership certificate needs to contain a member number and a member grade. A digital driver license should contain birthday and other information. Parameterized roles can also represent access permissions that take parameters identifying resources and access modes.

2.4 Common vocabularies

When an entity A defines $A.R$ to contain $B.R_1$, it needs to understand what B means by the role name R_1. This is the problem of establishing a common vocabulary. Different entities need a common vocabulary before they can use each others' roles. Common vocabulary is particularly critical in systems that support attribute-based delegations. For instance, the expression `EPub.university.stuID` only makes sense when universities use `stuID` for the same purpose.

In RT, we address this problem through a scheme inspired by XML namespaces [5]. We introduce *application domain specification documents (ADSDs)*. Each ADSD defines a vocabulary, which is a suite of related data types, role identifiers (role ids for short) with the name and the data type of each of its parameters, *etc.* An ADSD may also declare other common characteristics of role ids, such as storage type information, as studied in [20]. An ADSD generally should give natural-language explanations of these role ids, including the conditions under which credentials defining them should be issued. Credentials contain a preamble in which vocabulary identifiers are defined to refer to a particular ADSD, *e.g.*, by giving its URI. Each use of a role id inside the credential then incorporates such a vocabulary identifier as a prefix. Thus, a relatively short role id specifies a globally unique role id. An ADSD can refer to other ADSDs and use data types defined in it, using the mechanism just described. A concrete RT system is defined by multiple ADSDs.

Each ADSD defines a vocabulary. The notion of vocabularies is related to the notion of localized name spaces.

They both address name-space issues; however, they address issues at different levels. The notion of localized name spaces concerns itself with who has the authority to define the members of a role. The notion of vocabularies is about establishing common understandings of role names. For example, an accrediting board might issue an ADSD that defines the format of student ID credentials. Then a university can use this ADSD to issue student ID credentials. The university is still the authority of its own name space; it just uses the vocabulary created by another entity. The university can also freely choose which ADSD to use when issuing credentials. In particular, when there are multiple ADSDs about student IDs, the university can issue multiple credentials using different ADSDs.

2.5 Logical objects

In RT, roles are also used to represent permissions. A permission typically consists of an access mode and an object. It is often useful to group logically related objects and access modes together and to give permissions about them together. To do this, we introduce RT_2, which extends RT_1 with the notion of o-sets, which group logically related objects together. O-sets are defined in ways similar to roles. The difference is that the members of o-sets are objects that are not entities. RT_2 will be introduced in section 4.

2.6 Threshold and separation-of-duty policies

Threshold structures, which require agreement among k out of a list of entities, are common in trust-management systems. Some systems, such as Delegation Logic, also have the more expressive dynamic threshold structures, which are satisfied by the agreement of k out of a set of entities that satisfy a specified condition.

A related yet distinct policy concept is separation of duty (SoD) [7, 24]. This security principle requires that two or more different people be responsible for the completion of a sensitive task, such as ordering and paying for a purchase. SoD can be used to discourage fraud by requiring collusion among entities to commit fraud. In RBAC, SoD is often achieved by using constraints such as mutual exclusion among roles [23, 24] and requiring cooperation of mutually exclusive roles to complete sensitive tasks. Because no entity is allowed to simultaneously occupy two mutually exclusive roles, sensitive tasks can be completed only by cooperation of entities. This is sufficient, though not necessary, to ensure that cooperation between two entities is required to perform the sensitive task.

Though related, the threshold structures of existing TM systems cannot generally be used to express SoD policies. Threshold structures can require agreement only of two different entities drawn from a single set, while SoD policies typically are concerned with agreement among members of two different sets. For similar reasons, mutually exclusive roles cannot be used to achieve thresholds either.

Constraints such as mutual exclusion of roles are non-monotonic in nature, *i.e.*, an entity cannot be a member of one role if it is a member of another role. To enforce such constraints, complete information about role memberships is needed. Since we allow only monotonic credentials in RT, we cannot use such constraints. Instead, we use what we call *manifold roles* to achieve thresholds and separation of duty. Similar to a role, which defines a set of entities, a manifold role defines a set of *entity collections*, each of which is a set of entities whose cooperation satisfies the manifold role. Manifold roles are defined by role expressions constructed using either of the two *role-product operators*: $\odot$ and $\otimes$.

The role expression

$$\overbrace{A.R \otimes A.R \otimes \cdots \otimes A.R}^{k}$$

represents the dynamic threshold structure that requires k (different) entities out of members of $A.R$. The role expression "$A.R_1 \otimes A.R_2$" represents the set of entity collections each of which has two different entities, one from $A.R_1$ and the other from $A.R_2$. This can be used to force cooperation to complete a sensitive task (the goal of SoD) without forcing roles to be mutually disjoint. This could permit important flexibility, particularly in small organizations where individuals may need to fulfill several roles. Such flexibility motivates mutual exclusion in role activations (also known as dynamic separation of duty) [24]. Also, because the constructs are monotonic, they allow SoD to be supported in a de-centralized framework, where role membership information may be partial or incomplete.

The operator $\odot$ can be used to implement policies such as the following: An action is allowed if it gets approval from two roles. This approval might come from one entity who is a member of both roles, or it might come from two different entities who are each members of one role. Manifold roles and role-product operators are part of RT^T, which will be introduced in section 5.

2.7 Delegation of role activations

Above we have discussed delegation of authority to define a role. Let us now consider delegation of the capacity to exercise one's membership in a role.

In many scenarios, an entity prefers not to exercise all his rights. An administrator often logs in as an ordinary user to perform ordinary tasks. In another example, a user is temporarily delegated certain access rights by his manager during his manager's absence. The user will often want to exercise only his customary rights, wishing to use his temporary rights only when explicitly working on his manager's behalf. This notion is related to the least privilege principle and is supported by many systems. In RBAC, it is supported by the notion of sessions. A user can selectively activate some of his eligible roles in a session.

A natural generalization of user-to-session delegation of role activations is process-to-process delegation of those role activations. The need for this is particularly acute in distributed computing environments. Imagine the scenario in which a user starts a session, activating some of his roles, and then issues a request. To fulfill this request, the session process starts a second process on behalf of the user, which invokes a third process, which is running on a different host, so as to access back-end services needed to complete the requested task. Each of these processes must be delegated the authority to act on the user's behalf, and the first two must pass that authority to the processes they initiate.

Process-to-process delegation of role activations is not supported by RBAC. While similar policy concepts were studied in [1], the logic given there is intractable. Existing trust-management (TM) systems do not support selective role activations or delegation of those activations — an entity implicitly uses all of its rights in every request it makes. In a system where the requester provides credentials to support its request, the requester could limit the rights being exercised by providing only a subset of its credentials. However, this depends on the architectural assumption that credentials are provided by the requester. Even with this assumption, mechanisms are still needed for specifying which rights to use in support of a request, and for enforcing such specifications.

We introduce RT^D to handle delegation of the capacity to exercise role memberships. RT^D adds the notion of delegation of role activations to the RT framework. Such a delegation takes the form $B_1 \xrightarrow{D \text{ as } A.R} B_2$, which means that B_1 delegates to B_2 the ability to act on behalf of D in D's capacity as a member of $A.R$. This one form of delegation can be used to express selective role activations, delegation of role activations, and access requests supported by a subset of the requesting entity's roles. RT^D will be introduced in section 6.

2.8 Summary of components of the RT framework

RT_0 was introduced in [20]. In this paper, we present four more components of RT: RT_1, RT_2, RT^T, and RT^D. Following is a brief summary of the features in these components.

- RT_0 supports localized authorities for roles, role hierarchies, delegation of authority over roles, attribute-based delegation of authority, and role intersections.
- RT_1 adds to RT_0 parameterized roles.
- RT_2 adds to RT_1 logical objects.
- RT^T provides manifold roles and role-product operators, which can express threshold and separation-of-duty policies.
- RT^D provides delegation of role activations, which can express selective use of capacities and delegation of these capacities.

RT^D and RT^T can be used, together or separately, with each of RT_0, RT_1, or RT_2. The resulting combinations are written RT_i, RT_i^D, RT_i^T, and RT_i^{DT} for $i = 0, 1, 2$.

An RT system consists of application domain specification documents (ADSDs), definition credentials, and optionally delegation credentials (if using RT^D).

3 RT_1: Defining Roles

In this section, we introduce RT_1. RT_1 credentials define roles.

3.1 Syntax of RT_1 credentials

The syntax we use in this paper is an abstract syntax designed for understanding the framework. The representation used in practice can take various forms, *e.g.*, XML.

An RT_1 *credential* has a head and a body. The *head* of a credential has the form $A.r(h_1, \ldots, h_n)$, in which A is an entity, and $r(h_1, \ldots, h_n)$ is a role name. For $r(h_1, \ldots, h_n)$ to be a role name requires that r is a role identifier, and that for each i in $1..n$, h_i is a data term having the type of the ith parameter of r. In RT_1, a data term is either a constant or a variable, with just one exception in the type-3 credential below. We say that a credential with the head $A.r(h_1, \ldots, h_n)$ *defines* the role $A.r(h_1, \ldots, h_n)$. (This choice of terminology is motivated by analogy to name definitions in SDSI, as well as to predicate definitions in logic programming.) Such a credential must be issued by A, and so we call A the *issuer* of this credential. In the following, we present four types of credentials, each having a different form of body corresponding to a different way of defining role membership.

- *Type-1*: $A.r(h_1, \ldots, h_n) \longleftarrow D$.

  A and D are (possibly the same) entities. This credential means that A defines D to be a member of A's $R = r(h_1, \ldots, h_n)$ role. In the attribute-based view, this credential can be read as D has the attribute $A.R$, or equivalently, A says that D has the attribute R.

- *Type-2*: $A.r(h_1, \ldots, h_n) \longleftarrow B.r_1(s_1, \ldots, s_m)$

  A and B are (possibly the same) entities, and $R = r(h_1, \ldots, h_n)$ and $R_1 = r_1(s_1, \ldots, s_m)$ are (possibly the same) role names.

This credential means that A defines its R role to include all members of B's R_1 role. In other words, A defines the role $B.R_1$ to be more powerful than $A.R$, in the sense that a member of $B.R_1$ is automatically a member of $A.R$ and thus can do anything that the role $A.R$ is authorized to do. The attribute-based reading of this credential is: If B says that an entity has the attribute R_1, then A says that it has the attribute R.

- *Type-3*:
  $A.r(h_1, \ldots, h_n) \longleftarrow A.r_1(t_1, \ldots, t_\ell).r_2(s_1, \ldots, s_m)$

  We call $A.r_1(t_1, \ldots, t_\ell).r_2(s_1, \ldots, s_m)$ a *linked role*. The attribute-based reading of this credential is: If A says that an entity B has the attribute $R_1 = r_1(t_1, \ldots, t_\ell)$, and B says that an entity D has the attribute $R_2 = r_1(s_1, \ldots, s_m)$, then A says that D has the attribute $R = r(h_1, \ldots, h_n)$. If R and R_2 are the same, A is delegating its authority over R_2 to anyone who A believes to have the attribute R_1. This is an attribute-based delegation: A identifies B as an authority on R_2 not by using (or knowing) B's identity, but by another attribute of B (*viz.*, R_1).

  The data terms in the first role name $r_1(t_1, \ldots, t_\ell)$ in the linked role, *i.e.*, $t_1, \ldots, t_\ell$, can be a special keyword "`this`". It has the predefined type entity. The meaning of it will be explained in section 3.3.

- *Type-4*: $A.R \longleftarrow B_1.R_1 \cap B_2.R_2 \cap \cdots \cap B_k.R_k$

  In this credential, k is an integer greater than 1. We call $B_1.R_1 \cap B_2.R_2 \cap \cdots \cap B_k.R_k$ an *intersection*.[2] This credential means that if an entity is a member of $B_1.R_1, B_2.R_2, \ldots$, and $B_k.R_k$, then it is also a member of $A.R$. The attribute-based reading of this credential is: A believes that anyone who has all the attributes $B_1.R_1, \ldots, B_k.R_k$ also has the attribute R.

A variable that appears in an RT_1 credential can be either named or anonymous. A *named variable* takes the form of a question mark "?" followed by an alpha-numeric string. If a variable appears only once in a credential, it does not need to have a name and can be anonymous. An *anonymous variable* is represented by the question mark alone. Note that two different appearances of "?" in a credential represent two distinct variables. When a variable occurs as a parameter to a role name, it is implicitly assigned to have the type of that parameter.

A variable may optionally have one or more constraints following its name, separated by a colon. Each constraint is given by a static value set from which possible values of the variable can be drawn. The syntax of static value sets is introduced in section 3.2.

We now introduce the notion of *well-formed credentials*. In any RT system, a credential that is not well-formed is ignored. An RT_1 credential is *well-formed* if all named variables are well-typed and safe. A named variable is *well-typed* if it has the same type across all appearances in one credential. Two types are the same if they have the same name. A variable is *safe* if it appears in the body. As will be seen in section 3.3, this safety requirement ensures that RT_1 credentials can be translated into safe Datalog rules, thus help ensures tractability of RT_1.

Example 2 A company Alpha allows the manager of an employee to evaluate an employee.

`Alpha.evaluatorOf(?Y)`$\longleftarrow$`Alpha.managerOf(?Y)`

This policy cannot be expressed in RT_0.

3.2 Data types in RT_1

RT_1 has the following data types.

- **Integer types**. An integer type is ordered. When declaring an integer type, one can restrict its values by optionally specifying four facets: a min (default is $-\infty$), a max (default is ∞), a step s (default is 1), and a base value t (default is 0). The legal values of this type include all integer value v's such that $v = t + ks$ for some integer k and that $min \leq v \leq max$.

- **Closed enumeration types**. The declaration of a closed enumeration type declares it as either ordered or unordered, and lists the allowed values of this type. An ordered type has a corresponding integer type; the default is $[1..n]$ for a type of n elements. The corresponding type can also be explicitly specified. The boolean type is a predefined unordered closed enumeration type.

- **Open enumeration types**. An open enumeration type is unordered. The allowed values of an open enumeration type are not given statically; instead, each constant that appears in a place that requires this type is a value of this type. The entity type is a predefined open enumeration type.

- **Float types**. A float type is ordered. Defining a float type is very similar to defining an integer type. One can optionally specify the four facets, but these facets now take float values: a min (default is $-\infty$), a max (default is ∞), a step s (default is 1.0), and a typical value t (default is 0.0). The legal values of this type include all values v's such that $v = t + ks$ for some integer k and that $min \leq v \leq max$.

[2] In [20], an intersection can also contain entities or linked roles. The restriction here does not change expressive power: one can always add additional intermediate roles.

- **Date and time types**. There are predefined types for date, time, *etc.*These types are ordered.

For each type, one can write *static value sets*, which can be used to constrain variables in credentials. A value set is said to be static if the values in it do not depend on credentials. By contrast, a role can be viewed as a dynamic value set of the entity type. A static value set of an ordered type τ is represented by a set of non-intersecting ranges $\{l_1..u_1, l_2..u_2, \ldots, l_n..u_n\}$, where l_i's and u_i's are values of τ. When $l_i = u_i$, it can be written as just l_i. A static value set for an unordered type takes the form $\{v_1, \ldots, v_n\}$, where v_i's are constants of this type. Note that testing whether a constant is in a static value set takes time at most linear in the representation size of the value set.

Example 3 A University StateU gives special privileges to graduates from the first four years of its operation, no matter which degree was conferred.

```
StateU.foundingAlumni ⟵
  StateU.diploma(?, ?Year:[1955..1958])
```

Here, `diploma` is a role identifier that takes two parameters, a degree and a year, and "?" is an anonymous variable.

3.3 Translation to logic rules and tractability

We now define a translation from each RT_1 credential to a logical rule. This translation serves both as a definition of the semantics and also as one possible implementation mechanism. In the output language, we use a special binary predicate $isMember$, which takes an entity and a role as arguments. We also use domain predicates: for each static value set V, a unary predicate p_V is introduced, in which $p_V(v)$ is true for each value $v \in V$. These domain predicates are used for translating constraints on variables into logical atoms. Credentials are translated as follows:

1. From $A.R \longleftarrow D$ to

   $isMember(D, A.R)$.

   $A.R$ can be viewed as a shorthand for $role(A, R)$.

2. From $A.R \longleftarrow B.R_1$ to

   $isMember(?z, A.R) \longleftarrow$
   $\quad isMember(?z, B.R_1),$
   $\quad [conditions].$

   In the above, $?z$ is a variable, which we call *the implicit variable*.

   The optional *conditions* part comes from the constraints on variables. For each static value set V used as a constraint on a variable $?x$ in the credential, the *conditions* part includes a logical atom $p_V(?x)$, which we call an *arithmetic atom*. We call each logical atom of $isMember$ a *relational atom*.

   In the rest of the paper, we often omit the optional *conditions* part in translation formulas. Remember that they need to be added when there are constraints on variables.

3. From $A.R \longleftarrow A.R_1.R_2$ to

   $isMember(?z, A.R) \longleftarrow$
   $\quad isMember(?x, A.R_1),$
   $\quad isMember(?z, ?x.R_2).$

   Recall that the keyword "`this`" can be used as a data term in R_1. Each appearance of "`this`" is translated to the implicit variable $?z$. See example 4 for use of `this`.

4. From $A.R \longleftarrow B_1.R_1 \cap B_2.R_2 \cap \cdots \cap B_k.R_k$ to

   $isMember(?z, A.R) \longleftarrow$
   $\quad isMember(?z, B_1.R_1),$
   $\quad \ldots,$
   $\quad isMember(?z, B_k.R_k).$

Local access control policies take the same form as credentials. Recall that in RT, permissions are also represented as roles. When an entity D submits a request *req*, and this request is governed by the role $A.R$, the request should be authorized if $isMember(D, A.R)$ is provable from supporting credentials and policies. See [20] for work on collecting credentials when they are stored in a distributed fashion.

Example 4 As part of its annual review process, Alpha gives a pay raise to an employee if someone authorized to evaluate the employee says that his performance was good.

```
Alpha.payRaise ⟵
  Alpha.evaluatorOf(this).goodPerformance
```

Rules resulting from the above translation can be straightforwardly translated into Datalog by translating

$$isMember(?z, A.r(h_1, \ldots, h_n))$$

into

$$member(A, r, h_1, \ldots, h_n, ?z).$$

Given a set of RT_1 credentials $\mathcal{C}$, let $Trans(\mathcal{C})$ be the Datalog program resulting from the translation. The *implications* of $\mathcal{C}$, defined as the set of membership relationships implied by $\mathcal{C}$, is determined by the minimal model of $Trans(\mathcal{C})$. In the following, we show that RT_1 is tractable.

Proposition 1 *Given a set $\mathcal{C}$ of RT_1 credentials, assuming that each credential in $\mathcal{C}$ has at most v variables and that each role name has at most p arguments, then computing the implications of $\mathcal{C}$ can be done in time $O(MN^{v+2})$, where $N = \max(N_0, pN_0)$, N_0 is the number of credentials in $\mathcal{C}$, and M is the size of $\mathcal{C}$.*

Proof. An obvious evaluation algorithm is as follows. First compute $Trans(\mathcal{C})$. Then compute all ground instances of the resulting rules obtained by substituting variables by matching-type constants. The arithmetic atoms generated by constraints are evaluated during the instantiation process, and ground rules are thrown away if these constraints are not satisfied. Finally compute the model of the remaining set of ground rules. Since computing the minimal model of a set of ground Horn clauses can be done in linear time [9], the total time this process takes is linear in the size of the resulting ground program.

Consider the translation of one credential *cred*, let $Trans(cred)$ be the resulting rule. $Trans(cred)$ has up to v variables coming from *cred* and up to 2 variables introduced during the translation. For the variables from *cred*, the instantiation process considers only the (at most pN_0) constants that appear as parameters to role names in the heads of credentials in $\mathcal{C}$ because only these constants can appear as the c_i's in a ground atom $member(A, r, c_1, \ldots, c_q, D)$ in the minimal model of $Trans(\mathcal{C})$. This follows because each variable in the head of a credential must also appear in the body. If *cred* is a type-2, 3, or 4 credential, $Trans(cred)$ also has an implicit variable $?z$; to instantiate $?z$, only the (at most N_0) entities that appear on the right-hand sides of type-1 credentials in $\mathcal{C}$ need to be considered. If *cred* is a type-3 credential, then in addition to $?z$, $Trans(cred)$ also has another variable $?x$, which is instantiated only to the (at most N_0) entities that appear as the issuers of credentials in $\mathcal{C}$.

Therefore, the number of variables per rule is at most $v + 2$ after the translation, and there are $O(N = \max(N_0, pN_0))$ ways to instantiate each variable. For each rule, there are $O(N^{v+2})$ ways to instantiate it, and so the size of the ground program is $O(MN^{v+2})$. ∎

We argue that the variable bound v is typically bounded by $2p$, where p is the maximum arity of all role names in a vocabulary. A type-1 credential has no variable; a type-2 credential has at most p variables (because each variable in the head also appears in the body); a type-3 credential has at most $2p$ variables; and a type-4 credential has at most kp variables. The bound kp is reached when each of the intersecting roles contains an almost completely different set of variables,[3] which is rarely the case in practical policies. Also note that if one makes the restriction that each type-4 credential can have only 2 roles in the body, then the number of variables is bounded by $2p$.

[3]When the roles of an intersection can be partitioned into collections containing disjoint sets of variables, the credential can be broken up into several credentials with fewer variables per rule. However some intersections containing lots of variables cannot be broken up. An extreme case has p^2 variables arranged in a $p \times p$ matrix and $2p$ atoms, each atom contains a row or a column of the matrix. See [13] and the references in it for study on tractability of conjunctive queries.

Given a set of credentials $\mathcal{C}$, the time to answer a single request is clearly bounded by the time to compute all the implications of $\mathcal{C}$, which is polynomial is the size of $\mathcal{C}$. A trivial algorithm is to first compute the minimal model of $Trans(\mathcal{C})$ and check whether the request is true in the model. However, there are efficient ways to answer a query without computing the minimal model first. There has been extensive work in logic programming and deductive databases on how to answer queries more efficiently, *e.g.*, [22, 25].

In systems where the requester presents credentials to prove authorization, one might be concerned about the complexity of searching for conclusions of those credentials, and potential denial-of-service attacks. To combat this, the requester can be required to present a credential chain that is organized into a proof of authorization, where proof checking can be performed linearly.

4 RT_2: Describing Logical Rights

RT_2 adds to RT_1 the notion of *o-sets*, which are used to group logically related objects such as resources, access modes, *etc.* An o-set is formed by an entity followed by an o-set name, separated by a dot. An o-set name is formed by applying an *o-set identifier* (o-set id for short) to a tuple of data terms. An o-set id has a base type τ. O-set names and o-sets formed using an o-set id have the same base type as the o-set id. The value of an o-set is a set of values in τ.

An o-set-definition credential is similar to a role-definition credential. The head takes the form $A.o(h_1, \ldots, h_n)$, in which $o(h_1, \ldots, h_n)$ is an o-set name of base type τ. The body can be a value of base type τ, another o-set $B.o_1(s_1, \ldots, s_m)$ of base type τ, a linked o-set $A.r_1(t_1, \ldots, t_\ell).o_1(s_1, \ldots, s_m)$, in which $r_1(t_1, \ldots, t_\ell)$ is a role name and $o_1(s_1, \ldots, s_m)$ is an o-set name of base type τ, or an intersection of k o-sets of the base type τ.

A credential in RT_2 is either a role-definition credential or an o-set-definition credential. Credentials in RT_2 are more general that those in RT_1 in the following two aspects.

- A variable of type τ can be constrained by dynamic value sets of base type τ, *i.e.*, roles or o-sets.
- The safety requirement on variables is relaxed. A variable is safe if a) it appears in a role name or an o-set name that appears in the body of the credential; or, b) it is constrained by an o-set or a role; or c) it appears in a role or o-set that constrains a variable. As will be seen in section 4.1, this relaxed requirement suffices to guarantee tractability.

RT_2's extensions enable the following examples, which are not expressible in RT_1.

Example 5 Alpha allows members of a project team to read documents of this project:

```
Alpha.fileAc(read,?F:Alpha.documents(?proj))
  ⟵ Alpha.team(?proj)
```

The variable `?proj` is safe because it appears in the body, and the variable `?F` is safe because it is constrained by an o-set.

Given "`Alpha.documents(proj1)` ⟵ `fileA`" and "`Alpha.team(proj1)` ⟵ `Bob`", one can conclude that "`Alpha.fileAc(read, fileA)` ⟵ `Bob`".

Example 6 Alpha allows the manager of the owner of a file to access that file:

```
Alpha.read(?F)⟵
  Alpha.manager(?E:Alpha.owner(?F))
```

The variable `?E` is safe because it appears in the body, and the variable `?F` is safe because it appears in a role that constrains the variable `?E`.

Given "`Alpha.owner(file1)` ⟵ `userB`" and "`Alpha.manager(userB)` ⟵ `userC`", one can conclude that "`Alpha.read(file1)` ⟵ `userC`".

4.1 Translation to logic rules

An o-set-definition credential is translated into a logic rule in exactly the same way as a role-definition credential. We only need to extend the predicate $isMember$ to take o-sets and values of other types as arguments. For each constraint in which a variable $?x$ is constrained by $A.O$, add $isMember(?x, A.O)$ to the body of the rule.

Proposition 2 *RT_2 has the same computational complexity as RT_1. Given a set $\mathcal{C}$ of RT_2 credentials, computing the implications of $\mathcal{C}$ can be done in time $O(MN^{v+2})$.*

Proof. Because each constraint using a role or an o-set is translated into a relational atom in the body, the relaxed variable safety requirement suffices to guarantee that each variable in a rule appears in a relational atom in the body.

For any variable constrained by a role or an o-set, only the (at most N_0) same-type constants appearing on the right-hand-side of some type-1 credentials need to be used to instantiate the variable. The rest follows from the proof of Proposition 1. ∎

Note that an RT_2 rule can contain more variables than an RT_1 rule due to the use of o-sets and roles as constraints.

5 RT^T: Supporting Threshold and Separation-of-Duty Policies

One can express simple threshold structures by using intersections. For example, the policy that A says that an entity has the attribute R if two out of B_1, B_2, B_3 say so can be represented by three credentials $A.R \longleftarrow B_1.R \cap B_2.R$, $A.R \longleftarrow B_2.R \cap B_3.R$, and $A.R \longleftarrow B_3.R \cap B_1.R$. However, using intersections alone cannot express the policy that A says that an entity has attribute R if two *different* entities having attribute R_1 says so. Another potentially important policy allows something to happen if one entity of role $A.R_1$ and a different entity of role $A.R_2$ both request it. This is a common separation-of-duty policy; it cannot be expressed by the threshold structures in Delegation Logic or other previous trust-management systems.

To express policies like these, we introduce RT^T. Instead of introducing an operator just for thresholds, we introduce two more basic and more expressive operators. These can be used to implement threshold, separation of duty, and other policies.

More specifically, RT^T adds the notion of *manifold roles*, which generalizes the notion of roles. In contrast, we call the roles in RT_i *single-element roles*. A manifold role has a value that is a set of what we call *entity collections*. An entity collection is either an entity, which can be viewed as a singleton set, or a set of two or more entities. This allows us to view a single-element role as a special-case manifold role whose value is a set of singletons. In the rest of this paper, we extend the notion of *roles* to include both manifold roles and single-element roles, and we continue to use R to denote role name of this generalized notion of roles. RT^T introduces two new types of credentials:

- *Type-5*: $A.R \longleftarrow B_1.R_1 \odot \cdots \odot B_k.R_k$

  In which R and the R_i's are (single-element or manifold) role names. This credential means:

  $\mathit{members}(A.R) \supseteq \mathit{members}(B_1.R_1 \odot \cdots \odot B_k.R_k) = \{s_1 \cup \cdots \cup s_k \mid s_i \in \mathit{members}(B_i.R_i) \text{ for } 1 \le i \le k\}$.

  Here, when s_i is an individual entity, say, D, it is implicitly converted to the singleton $\{D\}$.

- *Type-6*: $A.R \longleftarrow B_1.R_1 \otimes \cdots \otimes B_k.R_k$

  This credential means:

  $\mathit{members}(A.R) \supseteq \mathit{members}(B_1.R_1 \otimes \cdots \otimes B_k.R_k) = \{s_1 \cup \cdots \cup s_k \mid (s_i \in \mathit{members}(B_i.R_i)\ \&\ s_i \cap s_j = \emptyset) \text{ for } 1 \le i \ne j \le k\}$

Example 7 A says that an entity has attribute R if one member of $A.R_1$ and two different members of $A.R_2$ all say so. This can be represented using the following credentials:

$A.R_3 \longleftarrow A.R_2 \otimes A.R_2, \quad A.R_4 \longleftarrow A.R_1 \odot A.R_3,$
$A.R \longleftarrow A.R_4.R.$

Suppose that in addition one has the following credentials:

$A.R_1 \longleftarrow B, \quad A.R_1 \longleftarrow E,$
$A.R_2 \longleftarrow B, \quad A.R_2 \longleftarrow C, \quad A.R_2 \longleftarrow D.$

Then one can conclude the following:

$members(A.R_1) \supseteq \{B, E\}$
$members(A.R_2) \supseteq \{B, C, D\}$
$members(A.R_3) \supseteq \{\{B, C\}, \{B, D\}, \{C, D\}\}$,
$members(A.R_4) \supseteq \{\{B, C\}, \{B, D\}, \{B, C, D\}, \{B, C, E\}, \{B, D, E\}, \{C, D, E\}\}$.

Now suppose one further has the following credentials:

$B.R \longleftarrow B,\ \ B.R \longleftarrow C,$
$C.R \longleftarrow C,\ \ C.R \longleftarrow D,\ \ C.R \longleftarrow E,$
$D.R \longleftarrow D,\ \ D.R \longleftarrow E,$
$E.R \longleftarrow E.$

Then one can conclude that $members(A.R) \supseteq \{C, E\}$, but one cannot conclude $members(A.R) \supseteq \{B\}$ or $members(A.R) \supseteq \{D\}$.

As noted in section 2.6, the $\otimes$ operator can be used to enforce separation of duty (SoD) without requiring mutual exclusion of roles. See section 6.2 for additional examples of SoD.

In RT^T, type 1 through 4 credentials are also generalized in that a manifold role name can appear where a role name is allowed, except when as a constraint to a variable.

Each role identifier has a *size*. The size of a manifold role id should be specified when the role id is declared in an ADSD. A single-element role id always has size one. A role name $r(t_1, \ldots, t_h)$ has the same size as r, and we have $size(A.R) = size(R)$. This size of a role limits the maximum size of each of its member entity set. For example, if $size(A.R) = 2$, then $members(A.R)$ can never contain $\{B_1, B_2, B_3\}$.

For an RT^T role-definition credential to be well-formed, it has to satisfy the additional requirement that the size of its head is always greater than or equal to the size of its body. And the size of its body is defined as follows:

$$\begin{cases} size(D) = 1 \\ size(A.R_1.R_2) = size(R_2) \\ size(B_1.R_1 \cap \cdots \cap B_k.R_k) = \max_{i=1..k} size(R_i) \\ size(B_1.R_1 \odot \cdots \odot B_k.R_k) = \sum_{i=1..k} size(R_i) \\ size(B_1.R_1 \otimes \cdots \otimes B_k.R_k) = \sum_{i=1..k} size(R_i) \end{cases}$$

5.1 Translation into logic rules

We extend the predicate *isMember* in the output language to allow the first argument to be an entity collection, and to allow the second argument to be a manifold role as well as a single-element role. Let t be the maximum size of all manifold roles in the system, we also introduce $2(t-1)$ new predicates set_k and $niset_k$ for $k = 2..t$. Each set_k takes $k+1$ entity collections as arguments, and $set_k(s, s_1, \ldots, s_k)$ is true if and only if $s = s_1 \cup \cdots \cup s_k$; where when s_i is an entity, it is treated as a single-element set. Each $niset_k$ is similar to set_k, except that $niset_k(s, s_1, \ldots, s_k)$ is true if and only if $s = s_1 \cup \cdots \cup s_k$ and for any $1 \leq i \neq j \leq k$, $s_i \cap s_j = \emptyset$.

The translation for type 1, 2, and 4 credentials is the same as that in section 3.3. The other three types are translated as follows:

- From $A.R \longleftarrow A.R_1.R_2$, when $size(R_1) = 1$, to

  $isMember(?z, A.R) \longleftarrow$
  $\quad isMember(?x, A.R_1),$
  $\quad isMember(?z, ?x.R_2).$

  when $size(R_1) = k > 1$, to

  $isMember(?z, A.R) \longleftarrow$
  $\quad isMember(?x, A.R_1),$
  $\quad isMember(?z, ?x_1.R_2),$
  $\quad \ldots,$
  $\quad isMember(?z, ?x_k.R_2),$
  $\quad set_k(?x, ?x_1, \ldots, ?x_k).$

- From $A.R \longleftarrow B_1.R_1 \odot \cdots \odot B_k.R_k$ to

  $isMember(?z, A.R) \longleftarrow$
  $\quad isMember(?z_1, B_1.R_1),$
  $\quad \ldots,$
  $\quad isMember(?z_k, B_k.R_k),$
  $\quad set_k(?z, ?z_1, \ldots, ?z_k).$

- From $A.R \longleftarrow B_1.R_1 \otimes \cdots \otimes B_k.R_k$ to

  $isMember(?z, A.R) \longleftarrow$
  $\quad isMember(?z_1, B_1.R_1),$
  $\quad \ldots,$
  $\quad isMember(?z_k, B_k.R_k),$
  $\quad niset_k(?z, ?z_1, \ldots, ?z_k).$

It is easy to see that this translation is an extension to that in section 3.3. When a credential contains no manifold roles, the resulting rule is the same.

Proposition 3 *Given a set $\mathcal{C}$ of RT^T credentials, let t be the maximal size of all roles in $\mathcal{C}$. Computing the implications of $\mathcal{C}$ can be done in time $O(MN^{v+2t})$.*

Proof. The resulting rules have atoms like $set_k(z, z_1, \ldots, z_k)$ in the body; these atoms are evaluated and removed during the grounding process, similar to arithmetic atoms generated from constraints. Consider a rule translated from a type-5 credential *cred*, the translation introduces new variables $?z, ?z_1, \ldots, ?z_k$. When the values of variables $?z_1, \ldots, ?z_k$ are fixed, the value of $?z$ is uniquely determined by $set_k(?z, ?z_1, \ldots, ?z_k)$. Given N entities, there are $O(N^s)$ entity collections of size $\leq s$. And so for each $i = 1..k$, there are $O(N^{size(R_i)})$ ways to instantiate $?z_i$. Therefore, there are $O(N^{size(R_1)+\cdots+size(R_k)}) = O(N^{size(R)}) = O(N^t)$ ways to instantiate $?z_1, \ldots, ?z_k$. The variables coming from *cred* (there are at most v of them) can be instantiated in $O(N^v)$ ways. So all together, the rule can be instantiated in $O(N^{v+t})$ ways. Similar arguments apply to type-6 credentials.

Consider a rule translated from a type-3 credential, the translation introduces variables $?z, ?x, ?x_1, \ldots, ?x_k$. For each of $?x_1, \ldots, ?x_k$, only the $O(N)$ entities that are issuers of credentials in $\mathcal{C}$ need to be considered. And when $?x_1, \ldots, ?x_k$ are fixed, $?x$ is uniquely determined by $set_k(?x, ?x_1, \ldots, ?x_k)$. Since $k = size(R_1) \leq t$, there are $O(N^t)$ ways to instantiate $?x, ?x_1, \ldots, ?x_k$. And there are $O(N^{size(R_2)}) = O(N^t)$ ways to instantiate z. So all together, a type-3 credential can be instantiated in $O(N^{v+2t})$ ways.

It is not hard to see that a type-1 credential can be instantiated in $O(N^v)$ ways; a type-2 or type-4 credential can be instantiated in $O(N^{v+t})$ ways; and so the complexity result follows. ∎

6 RT^D: Supporting Delegation of Role Activations

As discussed in section 2.7, RT^D has the notion of delegation of role activations, which can be used to express user-to-session and process-to-process delegation of capacity. For example, that an entity D activates the role $A.R$ to use in a session B_0 can be represented by a *delegation credential*, "$D \xrightarrow{D \text{ as } A.R} B_0$". We call "$D$ as $A.R$" a role activation. B_0 can further delegate this role activation to B_1 by issuing the credential, "$B_0 \xrightarrow{D \text{ as } A.R} B_1$". An entity can issue multiple delegation credentials to another entity. Also, several role activations can be delegated in one delegation credential. This is viewed as a shorthand for multiple delegation credentials.

A delegation credential can also contains a keyword "`all`". For example, "$B_0 \xrightarrow{\texttt{all}} B_1$" means that B_0 is delegating all role activations it has to B_1; and "$B_0 \xrightarrow{D \text{ as } \texttt{all}} B_1$" means that B_0 is delegating to B_1 those of B_0's role activations in which D is activating the roles.

A request in RT^D is represented by a delegation credential that delegates from the requester to the request. For example, that B_1 requests to read `fileA` in the capacity of "D as $A.R$" can be represented by: $B_1 \xrightarrow{D \text{ as } A.R}$ `fileAccess(read, fileA)`. Note that `fileAccess(read, fileA)` is not an entity. This delegation should be interpreted as being from B_1 to a dummy entity representing the request `fileAccess(read, fileA)`. An RT^D system assigns a unique dummy entity to each request. That B_1 is making the request *req* using all its capacities is represented by $B_1 \xrightarrow{\texttt{all}} req$.

Delegation of role activations is delegation of the capacity to act in a role. It is a different kind of delegation from delegation of authority to define a role, as in a role-definition credential "$A.R \longleftarrow B.R$". These differences will be further discussed in the following section.

6.1 Translation to logic rules

Now, let us consider what credentials (both definition credentials and delegation credentials, which include requests) mean in RT^D. Again, we do this by presenting a translation to logic rules. We introduce the predicate *forRole*. An logical atom $forRole(B, D, A.R)$ reads B is acting for "D as $A.R$"; it means that B is acting for the role activation in which D activates $A.R$. The atom $forRole(B, D, A.R)$ is true when D is a member of $A.R$ and D delegates this role activation through a chain to B. Note that D has to be a member of $A.R$ for $forRole(B, D, A.R)$ to be true. That D is a member of $A.R$ is equivalent to $forRole(D, D, A.R)$.

Being delegated the capacity to act in a role is strictly weaker than being a member of the role. The former is represented by $forRole(B, ?y, A.R)$ and the latter is represented by $forRole(B, B, A.R)$. This difference is most clearly shown in the translation of type-3 credentials below.

The translation described below subsumes the translations in section 5.1 for role definition credentials. Translation for O-set definition credentials remain unchanged.

- A delegation credential $B_1 \xrightarrow{D \text{ as } A.R} B_2$ is translated to:

  $forRole(B_2, D, A.R) \longleftarrow$
  $\quad forRole(B_1, D, A.R).$

  This rule means that B_2 is acting for "D as $A.R$" if B_1 is doing so. Note that D is the source of the capacity to act in the role $A.R$, and the entity D is always explicitly maintained during the inferencing.

- A delegation credential $B_1 \xrightarrow{D \text{ as } \texttt{all}} B_2$ is translated to:

  $forRole(B_2, D, ?r) \longleftarrow$
  $\quad forRole(B_1, D, ?r).$

  This rule means that if B_1 is acting for D activating a role (any role), then B_2 is doing so as well.

- A delegation credential $B_1 \xrightarrow{\texttt{all}} B_2$ is translated to:

  $forRole(B_2, ?y, ?r) \longleftarrow$
  $\quad forRole(B_1, ?y, ?r).$

  This rule means that if B_1 is acting for an entity (any entity) activating a role (any role), then B_2 is also doing so. This is a delegation of all capacities.

- A request is translated the same way as a delegation credential; the request is replaced by the dummy entity corresponding to it. For example, $B \xrightarrow{D \text{ as } \texttt{all}} req$ is translated to:

$$forRole(ReqID, D, ?r) \longleftarrow$$
$$forRole(B, D, ?r).$$

Here, $ReqID$ is the dummy entity for the request req.

If a request req is governed by the role $A.R$, and $ReqID$ is the dummy entity for req, then the request req is authorized if $forRole(reqID, ?y, A.R)$ is true. Note that the above authorization query has a variable $?y$. This means that on whose behalf this request is issued does not affect whether the request should be authorized. However, this variable will be instantiated by the authorizing chain, and the resulting information can be used in auditing. So, while delegation is not directly controlled, it can be regulated through auditing mechanisms.

- From $A.R \longleftarrow D$ to

  $$forRole(D, D, A.R).$$

  This rule means that D is acting for itself as $A.R$, *i.e.*, D is a member of $A.R$.

- From $A.R \longleftarrow B.R_1$ to

  $$forRole(?z, ?y, A.R) \longleftarrow$$
  $$forRole(?z, ?y, B.R_1).$$

  This rule means that anyone who is acting for "$?y$ as $B.R_1$" is also acting for "$?y$ as $A.R$", *i.e.*, activating $B.R_1$ implies activating $A.R$ because $B.R_1$ is more powerful than $A.R$. Note that this rule subsumes $forRole(?z, ?z, A.R) \longleftarrow forRole(?z, ?z, B.R_1)$, which means that any member of $B.R_1$ is also a member of $A.R$.

- From $A.R \longleftarrow A.R_1.R_2$,

  when $size(R_1) = 1$, to

  $$forRole(?z, ?y, A.R) \longleftarrow$$
  $$forRole(?x, ?x, A.R_1),$$
  $$forRole(?z, ?y, ?x.R_2).$$

  This rule means that when $?x$ is a member of $A.R_1$, activating $B.R_2$ implies activating $A.R$. Note that $?x$ has to be a member of $A.R_1$; it is insufficient if $?x$ is just delegated the capacity to act in the role $A.R_1$. This is because the credential $A.R \longleftarrow A.R_1.R_2$ implies $A.R \longleftarrow B.R_2$ only when B is a member of $A.R_1$.

  when $size(R_1) = k > 1$, to

  $$forRole(?z, ?y, A.R) \longleftarrow$$
  $$forRole(?x, ?x, A.R_1),$$
  $$forRole(?z, ?y, ?x_1.R_2),$$
  $$\cdots,$$
  $$forRole(?z, ?y, ?x_k.R_2),$$
  $$set_k(?x, ?x_1, \ldots, ?x_k).$$

  When an entity is a member of a role, it implicitly has the capacity to act in the role. However, the converse does not hold. In particular, when one is delegated the capacity to act to in a role, although one can access resources by using this capacity, one cannot use this capacity to affect the meanings of the definition credentials it issued. Role memberships are determined only by definition credentials and are not affected by delegation credentials. Thus, delegated capacity provides no additional authority to define roles.

- From $A.R \longleftarrow B_1.R_1 \cap \cdots \cap B_k.R_k$ to

  $$forRole(?z, ?y, A.R) \longleftarrow$$
  $$forRole(?z, ?y, B_1.R_1),$$
  $$\cdots,$$
  $$forRole(?z, ?y, B_k.R_k).$$

- From $A.R \longleftarrow B_1.R_1 \odot \cdots \odot B_k.R_k$ to

  $$forRole(?z, ?y, A.R) \longleftarrow$$
  $$forRole(?z, ?y_1, B_1.R_1),$$
  $$\cdots,$$
  $$forRole(?z, ?y_k, B_k.R_k),$$
  $$set_k(?y, ?y_1, \ldots, ?y_k).$$

- From $A.R \longleftarrow B_1.R_1 \otimes \cdots \otimes B_k.R_k$ to

  $$forRole(?z, ?y, A.R) \longleftarrow$$
  $$forRole(?z, ?y_1, B_1.R_1),$$
  $$\cdots,$$
  $$forRole(?z, ?y_k, B_k.R_k),$$
  $$niset(?y, ?y_1, \ldots, ?y_k).$$

In addition to being strictly weaker, delegation of role activations is also intended to be different from delegation of authority in that it is more dynamic, in the following sense. First, delegation credentials are not stored in a distributed fashion and searched as are definition credentials. Instead, each entity keeps the chain of delegation credentials ending at itself, and passes this chain when delegating to other entities. Second, delegation credentials typically accompany a request (which itself is represented by a delegation credential) and are processed and used when processing the request. As a contrast, definition credentials are often preprocessed and stored. Third, delegation credentials typically have a shorter valid life time than definition credentials.

Proposition 4 *Adding RT^D adds a factor of $O(N)$ to the complexity.*

Proof. For the rules resulting from definition credentials, there is an additional variable of type entity per rule. This adds an additional $O(N)$ factor.

We need to be careful when instantiating rules resulting from delegation credentials that involve the keyword `all`. For example, $B_1 \xrightarrow{\texttt{all}} B_2$ is translated to

"$\mathit{forRole}(B_2, ?y, ?r) \longleftarrow \mathit{forRole}(B_1, ?y, ?r)$". Rules that have the variable $?r$ should be instantiated last. We now show that they do not increase the complexity bound.

Consider how many ways one can instantiate the clause "$\mathit{forRole}(B_2, ?y, ?r) \longleftarrow \mathit{forRole}(B_1, ?y, ?r)$," which is the most expensive kind of clauses resulting from delegation credentials. Suppose that we are not using RT^T. The variable $?y$ needs to be instantiated with $O(N)$ entities, and $?r$ needs to be instantiated with the roles that appear in the head of some ground rules, for which there are at most $O(N^{v+1})$. So the clause needs to be instantiated in $O(N^{v+2})$ ways, which is the same as that of clauses resulting from type-3 credentials. When we are using RT^T, then $?y$ can be instantiated in $O(N^t)$ ways, where t is the maximum size of all roles, and the number of ways to instantiate $?r$ is still $O(N^{v+1})$, and so the total number of ways to instantiate is $O(N^{v+1+t}) = O(N^{v+2t})$. ∎

6.2 Examples using RT_1^{DT}

Example 8 In a small organization `SOrg`, any purchasing order has to be submitted and approved before it is placed. Any employee can submit a purchasing order. A manager can approve an order. A manager is also an employee; however, a manager cannot approve his own order. This can be represented as follows:

$\mathtt{SOrg.place} \longleftarrow \mathtt{SOrg.submit} \otimes \mathtt{SOrg.approve}$
$\mathtt{SOrg.submit} \longleftarrow \mathtt{SOrg.employee}$
$\mathtt{SOrg.approve} \longleftarrow \mathtt{SOrg.manager}$
$\mathtt{SOrg.employee} \longleftarrow \mathtt{SOrg.manager}$

Suppose that both `Alice` and `Bob` and managers:

$\mathtt{SOrg.manager} \longleftarrow \mathtt{Alice}$
$\mathtt{SOrg.manager} \longleftarrow \mathtt{Bob}$

`Alice` can submit an order by issuing:

$$\mathtt{Alice} \xrightarrow{\mathtt{Alice\ as\ SOrg.employee}} \mathtt{order(orderID)}$$

And `Bob` can approve it by issuing:

$$\mathtt{Bob} \xrightarrow{\mathtt{Bob\ as\ SOrg.approve}} \mathtt{order(orderID)}$$

Then one can prove that

$\mathit{forRole}(\mathtt{ReqID}, \{\mathtt{Alice}, \mathtt{Bob}\}, \mathtt{SOrg.place})$,

where `ReqID` is the dummy principal representing `order(orderID)`.

If `Bob` does not issue the above approval and `Alice` approves the order by also issuing:

$$\mathtt{Alice} \xrightarrow{\mathtt{Alice\ as\ SOrg.approve}} \mathtt{order(orderID)}$$

One still cannot prove that

$\mathit{forRole}(\mathtt{ReqID}, \{\mathtt{Alice}, \mathtt{Bob}\}, \mathtt{SOrg.place})$.

Now consider another example, which is from [1]. It is not exactly the same, because the logic in [1] (which we will call the ABLP logic) has both groups and roles and uses roles differently. However, it captures the intended scenarios and policies. This example is expressed in RT straightforwardly.

Example 9 A server `S` authorizes `fileA` to be deleted if it is requested from a good workstation on behalf of a user. `S` knows that `alice` is a user and trusts `CA` in certifying public keys for users. `S` knows that `ws1` is a good workstation and trusts `CA` in certifying public keys for workstations. These are expressed in the following credentials:

$\mathtt{S.del(fileA)} \longleftarrow \mathtt{S.user} \otimes \mathtt{S.goodWS}$
$\mathtt{S.user} \longleftarrow \mathtt{CA.userCert(alice)}$
$\mathtt{S.goodWS} \longleftarrow \mathtt{CA.machineCert(ws1)}$

The following are credentials issued by `CA`:

$\mathtt{CA.userCert(alice)} \longleftarrow \mathtt{K\_alice}$
$\mathtt{CA.machineCert(ws1)} \longleftarrow \mathtt{K\_ws1}$

In the setting studied in [1], a work station stores its private key in tamper-resistant firmware. When it boots, it generates a key pair for the operating system and issues a credential to delegate the activation of `S.goodWS` to the new key. When the user `alice` logs into a workstation `ws1`, a new process `p1` is set up and a new key pair is generated. Through `p1`, `alice` then makes a request to the server `S` to delete `fileA`. The process `p1` sets up a secure channel `Ch` to the server, and then sends the request through the channel. The following are delegation credentials that are needed.

$\mathtt{K\_ws1} \xrightarrow{\mathtt{K\_ws1\ as\ S.goodWS}} \mathtt{K\_os1}$
$\mathtt{K\_os1} \xrightarrow{\mathtt{K\_ws1\ as\ S.goodWS}} \mathtt{K\_p1}$
$\mathtt{K\_alice} \xrightarrow{\mathtt{K\_alice\ as\ S.user}} \mathtt{K\_p1}$
$\mathtt{K\_p1} \xrightarrow{\mathtt{K\_ws1\ as\ S.goodWS,\ K\_alice\ as\ S.user}} \mathtt{K\_Ch}$

The request sent by `K_Ch` to delete `fileA` on behalf of user alice working on a good workstation is represented as:

$$\mathtt{K\_Ch} \xrightarrow{\mathtt{K\_ws1\ as\ S.goodWS,\ K\_alice\ as\ S.user}} \mathtt{del(fileA)}.$$

And this request should be authorized.

7 Discussions, Future, and Related Work

We have implemented and used RT_0, and we are in the process of implementing other components of the RT framework. The RT_0 inference engine is implemented in Java, using algorithms described in [20]. We have constructed two demonstration applications using RT_0, a distributed scheduling (calendar) system and a web-based file sharing system. They are implemented by translating policies from a user interface into RT_0. Policies in these two applications often require features outside of RT_0. These needs drove the development of the features reported in this paper, as well as additional features discussed in section 7.2. Currently, policies using features outside RT_0 are handled by adding an ad-hoc layer on top of RT_0. We are in the process of extending the algorithms in [20] to other, more expressive, components in the framework and implementing them. We will then use them to replace the ad-hoc layer.

In general, credentials may contain sensitive information. To protect sensitive credentials while allowing them to be used in a decentralized environment, ABAC can be applied to credentials, as to any other resource. Trust can be established between two entities in such a context through an iterative process of revealing credentials to one another, called a trust negotiation [27]. Concurrently with the design of RT, we are developing a system for automated trust negotiation that supports RT. A design supporting RT_0 is presented in [26], where additional references to work in this area can also be found.

7.1 Implementation

A straightforward approach to implement an inference engine for RT is to use the translation process described in this paper together with a Datalog inference engine. However, there are several drawbacks of this approach. First, most Datalog or Prolog engines do not suit our needs. To work with a large number of credentials stored in a distributed way, we require a goal-oriented (top down) inference engine. This rules out many deductive database implementations that use bottom-up evaluation algorithms. We also require the inference engine to guarantee termination. This rules out many Prolog engines, which run into infinite loops with recursive rules. Second, the size of a full-blown Datalog inference engine might be unacceptable for some applications. For instance, XSB [14] is a system that satisfies our goal-orientation and termination requirements. In fact, one of the authors has used XSB previously while working on Delegation Logic [17]. Unfortunately, even the stripped down version of the XSB is several megabytes, while the jar file of the current RT_0 engine is less than 40KB. Third, based on our experience, it is often hard to integrate with a Datalog engine closely; and one has less control than needed during the inference process. For example, it might be hard to interleave credential collection with inferencing steps, as needed for some applications. Last but not least, we require support for functions and predicates for application-defined data types. For these reasons, we are developing direct algorithms, which can be implemented in general programming languages.

We use XML to represent both ADSDs and credentials. Credentials are digitally signed. We plan to address the revocation issue by requiring that each credential have an issue time and a validity period, similar to the approach suggested in [18]. Delegation credentials should have short life time and be renewed when needed. Definition credentials can have verification mechanism descriptions. For example, a verification mechanism might include an address for finding a credential revocation list (CRL), or the address and public key of an online certificate status verification server.

7.2 Future Work

In this paper, we have restricted our design to features that can be implemented in safe Datalog. Several further features that cannot be implemented in Datalog would also be desirable, however. While non-monotonicity is typically inappropriate for a de-centralized environment, certain non-monotonic constraints may be useful and appropriate. For example, mutual exclusion among roles that can be activated at the same time (also known as dynamic separation of duty) may be important to some organizations. Enforcing this policy requires complete information only about which roles are being activated, which can be expected to be available when processing access requests. A second desirable feature would enable authorization to depend on state information, such as history or environment data. For instance, history information is needed to implement the Chinese Wall policy [6]. Policies representing such policies often result in unsafe Datalog rules. We plan to address this by distinguishing a class of request-processing rules that are used only in connection with a specific request. When the request is made, it supplies values for all variables in the request-processing rule that might otherwise be unsafe. Yet another desirable feature is to be able to represent unbounded, structured resources, such as directory hierarchies. Handling these requires going beyond Datalog. We plan to address this by using Datalog with constraints to replace Datalog as the underlying foundation. Ongoing work on such extensions will be reported in near future.

7.3 Related Work

In section 1, we discussed the limitations of capability-style systems such as KeyNote and SPKI 1.0 and argued that the trust-management systems SPKI/SDSI [8, 10], KeyNote [2], and TPL [15] cannot express the five requirements for attribute-based access control. Here, we give additional comparisons of RT with related work.

RT unifies RBAC and trust-management concepts; it thus differs from previous TM systems in that it uses roles as a central notion. One advantage of this is the ability to allow selective role activation and delegation of these activations. This supports using partial authority in a request, which no previous TM systems support. In addition, the two role product operators in RT^D are more expressive than threshold structures in existing trust management systems.

The SDSI part of SPKI/SDSI is equivalent to RT_0 minus type-4 credentials (intersection). The SPKI part of SPKI/SDSI is a capability-style system. SPKI/SDSI is roughly equivalent to RT_1 with no variables, but it has one compound data type for tags.

KeyNote [2] is a capability-style system similar to SPKI. A KeyNote credential is similar to an RT credential of the form: "$A.r(h_1, \ldots, h_n) \longleftarrow B.r(h_1, \ldots, h_n), \mathit{conditions}$", where $\mathit{conditions}$ are boolean expressions on $h_1, \ldots, h_n$. However, KeyNote allows $\mathit{conditions}$ to contain operators like regular expression matching. This is fairly expressive, but has the disadvantage of being nondeclarative.

As we discussed in section 1, DL does not have the abstraction of subjects. On the other hand, DL has integer delegation depth and allows query of delegations; these are not allowed in RT. $RT_{0,1,2}$ can be viewed as syntactically sugared version of a subset of DL.

Our treatment of RT^D uses role activations like "D as $A.R$" and atoms like $\mathit{forRole}(B, D, A.R)$. The ABLP logic [1] also has "as", which it uses for restricting privileges. There, "D as R" has less privilege than D; furthermore, "D_1 as R" and "D_2 as R" may have different privileges. In RT, the use of roles follows that in RBAC, and "as" is interpreted as activating roles. In RT, "D_1 as $A.R$" has the same privilege as "D_2 as $A.R$". The difference between D_1 and D_2 becomes significant only when the role product operator $\otimes$ is involved and/or auditing is desired. The ABLP logic also has an operator "for", which can be encoded using a quoting operator $|$, and there is also a conjunction operator $\wedge$. The logic allows one to write arbitrarily long and complex principal expressions using these operators. The combination of $\wedge$ and $|$ makes the ABLP logic intractable, even though it does not support localized name space for roles or parameterized roles. In RT, $\wedge$ is implicitly achieved by having multiple credentials and manifold roles. The statement $\mathit{forRole}(B, D, A.R)$ in RT^D can be roughly read as "B for (D as $A.R$)"; and this is the only form of statement RT^D allows. Note that "B for B_1 for D as $A.R$" can be achieved by two delegation credentials from D to B_1 and then to B. We do not distinguish between "B for B_1 for D as $A.R$" and "B for B_2 for D as $A.R$". They are achieved by different credential chains; but B would have the same privilege in both cases. This reduces computational complexity of RT, makes it easier to understand, and still seems sufficient to capture policy concepts motivating the ABLP logic.

8 Conclusions

We introduce the RT framework, a family of Role-based Trust-management languages for representing policies and credentials in distributed authorization. RT combines the strengths of role-based access control and trust-management systems and is especially suitable for attribute-based access control. We present four components of the RT framework: RT_1, RT_2, RT^T, and RT^D. Together, they have seven forms of credentials and support localized authority of roles, delegation in role definition, linked roles, parameterized roles, manifold roles, and delegation of role activations. We also presented a translation from RT credentials to Datalog rules, which both serves as a logic-based semantics for RT and shows that the semantics is algorithmically tractable.

Acknowledgements

This work is supported by DARPA through SPAWAR contracts N66001-01-C-8005 and N66001-00-C-8015 and MURI grant N00014-97-1-0505 administrated by ONR. Raghuram Sri Sivalanka made some helpful comments on an earlier version of this paper. We also thank anonymous reviewers for their helpful reports.

References

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *Transactions on Programming Languages and Systems*, 15(4):706–734, October 1993.

[2] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system, version 2. IETF RFC 2704, September 1999.

[3] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.

[4] Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-7)*, pages 134–143. ACM Press, November 2000.

[5] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. W3C Recommendation, January 1999.

[6] David F.C. Brewer and Michael J. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–218, Los Alamitos, May 1989. IEEE Computer Society Press.

[7] David D. Clark and David R. Wilson. A comparision of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, May 1987.

[8] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[9] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[10] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. IETF RFC 2693, September 1999.

[11] Stephen Farrell and Russell Housley. An Internet attribute certificate profile for authorization, 2001.

[12] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 153–159, November 1997.

[13] Martin Groher, Thomas Schwentick, and Luc Segoufin. When is the evaluation of conjunctive queries tractable? In *Proceedings of the 33rd Annual Symposium on Theory of Computing (STOC'01)*, pages 657–666, July 2001.

[14] The XSB Research Group. The XSB programming system. `http://xsb.sourceforge.net/`.

[15] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 2–14. IEEE Computer Society Press, May 2000.

[16] Trevor Jim. SD3: a trust management system with certificate evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115. IEEE Computer Society Press, May 2001.

[17] Ninghui Li. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, September 2000.

[18] Ninghui Li and Joan Feigenbaum. Nonmonotonicity, user interfaces, and risk assessment in certificate revocation (position paper). In *Proceedings of the 5th Internation Conference on Financial Cryptography (FC'01)*. To be published by Springer. http://crypto.stanford.edu/~ninghui/papers/fc01.pdf.

[19] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. A practically implementable and tractable delegation logic. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 27–42. IEEE Computer Society Press, May 2000.

[20] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management (extended abstract). In *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8)*, pages 156–165. ACM Press, November 2001.

[21] Emil Lupu and Morris Sloman. Reconciling role based management and role based access control. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 135–141, November 1997.

[22] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–55, January 1999.

[23] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[24] Tichard T. Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *Proceedings of The 10th Computer Security Foundations Workshop (CSFW-10)*, pages 183–194. IEEE Computer Society Press, June 1997.

[25] Jefferey D. Ullman. *Principles of Databases and Knowledge-Base System*, volume 2. Computer Science Press, 1989.

[26] William H. Winsborough and Ninghui Li. Towards practical automated trust negotiation. To appear in *IEEE 3rd Intl. Workshop on Policies for Distributed Systems and Networks (Policy 2002)*. June 2002.

[27] William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*. IEEE Press, January 2000.