CERIAS Tech Report 2004-03

RTML: A ROLE-BASED TRUST-MANAGEMENT MARKUP LANGUAGE

Ninghui Li, John C. Mitchell

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

William H. Winsborough

Center for Secure Information Systems
George Mason University
4400 University Drive, Mail Stop 4A4
Fairfax, VA 22030-4444

Kent E. Seamons, Michael Halcrow, Jared Jacobson

Computer Science Department
Brigham Young University
Provo, UT 84602

RTML: A Role-based Trust-management Markup Language

Ninghui Li
Department of Computer Science
Purdue University
656 Oval Drive, West Lafayette, IN 47907-2096
ninghui@cs.purdue.edu

John C. Mitchell
Department of Computer Science
Stanford University
Gates 4B, Stanford, CA 94305-9045
jcm@stanford.edu

William H. Winsborough
Center for Secure Information Systems, George Mason University
4400 University Drive, Mail Stop 4A4 Fairfax, VA 22030-4444
wwinsborough@acm.org

Kent E. Seamons Michael Halcrow Jared Jacobson
Computer Science Department, Brigham Young University, Provo, UT 84602
{seamons, mhalcrow, jmj52}@cs.byu.edu

Abstract

RT is a framework for Role-based Trust Management [20]. In comparison with systems like SPKI/SDSI and KeyNote, the advantages of RT include: a declarative, logic-based semantic foundation, support for vocabulary agreement, strongly-typed credentials and policies, more flexible delegation structures, and more expressive support for Separation-of-Duty policies.

This paper describes advances in the RT framework that broaden its applicability and presents RTML, an XML-based data representation for RT policies and credentials. Improvements in RT include new data types to encode permissions involving structured resources and ranges, restrictive inheritance of roles for flexible refinement of permissions, and notions of identity roles and identity-based roles for enforcing separation-of-duty when a physical user holds multiple keys. RTML establishes a precise format for RT credentials and policies, facilitating deployment of the RT framework.

1 Introduction

RT is a framework for Role-based Trust Management. The framework comprises several sub-languages, described in previous papers [20, 21]. RT credentials define role membership and may delegate authority to add additional members to a role. RT's notion of role is more general than typically used in Role Based Access Control (RBAC) [23]. Roles in RT are localized to each principal, can have parameters, and can express concepts such as identity in systems like X.509 [14], role and permission in RBAC, name and authorization in SPKI/SDSI [11], and attribute in attribute certificates.

One distinguishing feature of the RT framework is that it directly addresses the issue of vocabulary agreement. When credential chains delegate access permissions of resources, all the principals involved in the chain need to use consistent terminology to specify resource permissions and delegation conditions. When different credential issuers use incompatible schemes, their credentials cannot be meaningfully combined. Some intended permissions may not be granted, or, when schemes intended for different purposes accidentally interact, unintended authorization may follow. Some systems do not address this issue at all; others try to come up with one vocabulary for all applications. Our philosophy is that, although different applications often share common policy concepts, they need to be able to use different vocabularies. In RT, we address this issue through a scheme inspired by XML namespaces [7]. We introduce *application domain specification documents (ADSDs)*. Each ADSD is globally uniquely identified by a URI, and defines a suite of related data types and role names, called a *vocabulary*.

Credentials, when using a role name, refer to the ADSD in which the role name is declared. This enables RT to have strongly typed credentials and policies. This feature helps ensure interoperability and reduce the possibility of errors in writing policies and credentials and unintended interaction of credentials.

In the process of turning the design of RT [20] into a system that can be used by several projects, we extended the design in the following ways.

First, two new categories of data types are added: *tree types* and *record types*. Tree types can be used to represent structured resources like file hierarchies, DNS names, etc. Record types can be used to group logically related fields together, e.g., an address may be defined using a record type that contains fields such as street number, zip code, etc. The credential safety requirements is also relaxed. These extensions enable one to represent permissions that involve structured resources and ranges, without sacrificing the tractability property.

Second, we add the notion of *restrictive inheritance* among roles to support flexible refinement of permissions. Consider the following example in SPKI. A firewall delegates to K_A the permission "(connect cs.stanford.edu)" and allows K_A to further delegate. K_A in turn delegates these rights to K_B. Now if K_B grants to K_C the permission "(connect cs.stanford.edu 80)", which is augmented with a port number, one can conclude that the firewall authorizes K_C to connect to the host through the given port. It is straightforward to achieve this in RT by encoding these permissions using a role with the name "connect" and two parameters: host and port. However, suppose the firewall administrator first declares the connect role to take the host parameter, alone, and later, after K_A has delegated the connect role to K_B, realizes the need for adding the port parameter. In this case we want to avoid requiring K_A to issue its delegation again, as is the case with the original design of RT. This is achieved by declaring another role, say, "connect2" to inherit "connect". (This is a restrictive form of inheritance, in that membership in connect2 implies authorization to fewer resources than does membership in connect.) This approach achieves the flexibility of untyped credentials without giving up the other advantages of strong typing.

Third, we now explicitly address the relationships between physical users and principals (keys). RT has manifold roles and exclusive product, designed to support Separation of Duty policies, which require that two or more different people be responsible for the completion of a sensitive task. This purpose could be defeated in the original RT design by a user possessing multiple keys. To address this issue, we introduce identity roles and identity-based roles.

In this paper, we also present RTML version 1, an XML-based data representation of the RT framework with the above described extensions. (We omit "version 1" in the rest of this paper.) This work on RTML, fleshing out the design of RT, involves efforts from three research projects, with focuses on trust management, attribute-based access control, and automated trust negotiation. Each of the projects found previous public-key certification systems such as X.509, SPKI/SDSI, and KeyNote [3] too limited in one way or another [20, 25, 27]. RTML's main purpose is to be used as an alternative to these systems.

A major design decision in RTML is how ADSDs and credentials interact. We explored the possibility of making ADSDs themselves XML schemas [12], but settled on making both ADSDs and credentials XML documents. One can think of ADSDs as `.h` files in C programs and credentials as `.c` files. Data types and role names are declared in ADSDs, and credentials must use these role names in a type-consistent way. The RTML system includes an XML schema document that defines the syntax of ADSDs and credentials.[1] The RTML system also has a predefined ADSD which includes data types for representing commonly used basic types, distinguished names, email addresses, DNS names, and file paths; this ADSD is provided in Appendix B.1.

1.1 Scenarios

In the following, we use three example scenarios to illustrate what can be expressed in RT. In these scenarios, we assume that public key certificates are used. None of the three scenarios can be fully captured using SPKI/SDSI, KeyNote, or attribute certificates. The latter two cannot be expressed in the original design of RT in [20] and require the new extensions introduced in this paper. We will show how to express these scenarios in Section 3.

Scenario 1 (Linking Delegation and Intersection) A fictitious Web publishing service, EPub, offers a discount to anyone who is both a graduate student and an ACM member since 2001. EPub delegates the authority over the identification of students to entities that EPub believes are legitimate universities. EPub additionally delegates the authority over identifying universities to a fictitious Accrediting Board for Universities, ABU. Bob is an ACM member and a master student of StateU, which is accredited by ABU.

EPub knows the public key of ACM and ABU. The public key of StateU, which issues Bob's student credential, should be certified in StateU's accrediting credential. Furthermore, EPub requires that the same university name appears in both its accreditation credential and the student's credential. Similarly, EPub requires that the same student name appears in both the ACM member credential and the student credential.

[1]The schema is available at the following URL: http://crypto.stanford.edu/~ninghui/rtml/RTMLv1.0q.xsd

Scenario 2 (Controlled Delegation of Permissions) A firewall FW issues a credential to give a system admin, SA, the authority to grant to anyone who has a Stanford ID permission to connect to the host "cs.stanford.edu" and to any host in the domain "cs.stanford.edu". The permission is expressed using a role with one parameter, "host". Later, FW wants to further parameterizes the permission by adding the port parameter. SA now grants to Alice, who has a Stanford ID, the permission to connect to any host in the domain "stanford.edu" at any port between 8000 and 8443. Then, when Alice requests to connect to the host "cs.stanford.edu" at port 8443, FW should allow this connection.

Note that SA can effectively delegate only to entities who have Stanford IDs. This is what we call a controlled delegation. In fact, when SA does not have a Stanford ID, he will not be able to get any connection permission by using the credential issued to him, although he can manage the permission.

Scenario 3 (Identity-based Separation of Duty) A bank FB has three roles: manager, cashier, and auditor. FB's policy requires that a certain transaction be approved by a manager, two cashiers, and an auditor. The two cashiers must be different. A manager who is also a cashier can serve as one of the two cashiers. And the auditor must be different from the other parties in the transaction.

The intention of the policy is to require different users, who are members of appropriate roles, to be jointly responsible for a transaction. A further complication arises when a physical user may possess multiple public keys. We want to ensure that the transaction is approved by distinct users, rather than distinct keys. To make this possible, we assume that FB assigns a unique employee number to each user, and every key that a user possesses is certified to be associated with the user's employee number.

1.2 Organization

The rest of the paper is organized as follows. Background information is given in Section 2. In Section 3, we show how to express the three scenarios described in Section 1.1 and explain the new extensions added to RT. In Section 4, we describe RTML. We then discuss related work in Section 5, and conclude in Section 6.

2 Background

In this section, we give background information on trust management and the RT framework.

2.1 Trust Management

The term "trust management" was introduced in [5] to group together several principles in dealing with authorization in decentralized distributed systems. Some of these principles appeared in earlier work on distributed access control, e.g., [1, 16]. The concept of trust management is later used and extended in several systems [3, 4, 6, 9, 11, 17, 20, 21].

In the "trust-management" approach, a requester submits a request to an *authorizer*, who specifies *access rules* (also called *policies*), which govern access to protected resources; then the authorizer and the requester jointly compute a set of credentials to be used in this request. Finally the authorizer decides whether to authorize this request by answering the *proof-of-compliance* question: "Do the access rules and credentials authorize the request?"

A TM system has a TM language for specifying credentials and access rules. A TM system also needs a process of computing which credentials to use in an authorization procedure; this is often called credential (or certificate) chain discovery [9]. When credentials are stored in a distributed manner, the discovery process needs to consider how to locate these credentials [21]. When credentials and/or access rules are considered sensitive, the discovery process may be carried out by using a "automated trust negotiation" process [24, 27, 28, 29].

A TM system also needs a proof-of-compliance checking program. A chain discovery program can also be used for proof-of-compliance, since if it can discover a chain, it certainly can verify that the chain is valid. However, a compliance checking program can often be simpler, since it is often not difficult to have the discovery program produce a chain that is organized in a way such that checking the chain is easier. These programs can be used as services independent of applications or embedded in applications.

2.2 History of the RT framework

The design of RT was influenced by systems like the logic for access control in [1, 16], SPKI/SDSI [11, 22], and Delegation Logic [17, 18]. The most basic part of RT, RT_0, was presented in [21], 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. A trust negotiation protocol that supports RT_0 credentials was introduced in [27]. Four additional components of the RT framework were introduced in [20]: RT_1, RT_2, RT^T, and RT^D. RT_1 adds to RT_0 parameterized roles. 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 separation-of-duty policies. RT^D provides delegation of role activations, which can express selective use of capacities and delegation of these capacities. In [19], Constraint DATALOG is used to extend

RT_1 with the ability to express permissions regarding structured resources and ranges, while at the same time ensuring tractability of evaluating implications of access rules and credentials.

In this paper, we describe RTML version 1, which implements constraint-enhanced RT_1^T in the RT framework[20]; it does not yet have RT_2 or RT^D. On the other hand, RTML extends the original design to address several practical issues, as discussed in Section 1.

In RTML, access rules are similar to credentials and only involve properties of requesters. This suffices for some applications. In other applications, access rules may include other conditions of access, such as current time, application state, auditing requirements, etc. RTML does not yet support these features. In those applications, RTML may still be used for expressing credentials.

2.3 Background of RT

A *principal*[2] can issue credentials and make requests. RT assumes that it can be verified that a principal indeed issued a particular credential or request. The most typical kind of principals are public keys, but RT does not mandate so. In some environments, a principal could also be, say, a symmetric key or a user account. In one web-based file sharing demonstration application we developed, user ids are used as principals.

The most important concept in RT is that of *roles*. Roles in RT are localized to principals. Each principal has its own authority name space for roles; this is the same as localized name spaces in SDSI. One needs to use a principal and a role term to refer to a role, e.g., $K_A.R$, in which K_A is a principal and R is a role term. In the simplest case, a role term just contains the name of role. More generally, a role term may contain parameters. A role $K_A.R$ can be read as K_A's R role. Only K_A has the authority to define the members of the role $K_A.R$, and K_A does so by issuing role-definition credentials. A role may be defined by multiple credentials; the effect is that of union.

RT has *single-element roles* and *manifold roles*. The semantics of a single-element role is a set of principals. The notion of single-element roles unifies several concepts in access control and trust management literature, including groups in many systems, identity in identity certification systems such as X.509, roles and permissions in RBAC, names in SDSI, authorization tags in SPKI, and attributes in attribute certificates. It is possible to unify these concepts because the common mathematical underpinning of the semantics of these concepts is sets of principals. A group is clearly a set of principals. An identity is a set of principals corresponding to one physical user; some systems require the set to contain just one principal. (The notion of identity and the relationship between users and principals will be further explored in Section 3.3.) A role in RBAC can be viewed as a set of principals who are members of this role; role hierarchy relationships can be viewed as ways to define role memberships.[3] A permission corresponds to a set of principals who have the permission. Granting a permission to a principal amounts to making the principal a member of the set corresponding to the permission. Granting a permission to a role amounts to asserting that the set corresponding to the permission includes as a subset the set corresponding to the role. A name in SDSI is also resolved to a set of principals. An attribute is a set of principals who have the attribute.

The notion of manifold roles generalizes that of single-element roles to allow each member of the role to be a principal set, instead of a principal. That the principal set $\{K_1, K_2\}$ is a member of the manifold role $K_A.R$ means that K_1 and K_2 together have the privileges associated with $K_A.R$, but either one of them acting alone may not have that privilege. The semantics of a manifold role is a set of principal sets. Manifold roles are introduced to support Separation of Duty policies [8, 26] in a more expressive way than do threshold structures.

3 Extensions to RT

In the following subsections, we present the new extensions to RT by describing how to express the policies associated with each of the three application scenarios given in Section 1.1.

3.1 Linking Delegation and Intersection

ABU, the fictitious accrediting board for universities, creates an ADSD, in which two roles are declared: university (which has only one parameter: name) and student (which has five parameters: university, department, program, id, and name). ACM creates an ADSD and declares one role: acmMember, which has four parameters: name, class, number, and since. EPub creates an ADSD for its own use, which declares a new role discount and includes the two ADSDs by ABU and ACM.

The credentials, access rules, and conclusions drawn from them, are given in Figure 1 in an abstract syntax. Line (1) represents the accrediting credential of

[2]Principals are called entities in earlier papers on RT [20, 21]; here we use "principal" to avoid potential confusion with entities in XML.

[3]This view of a role loses some of the meaning that can be associated with a role in RBAC, e.g., constraints. Constraints like mutually exclusive roles are implemented in RTML by using manifold roles. Constraints like cardinality constraints can be associated with the set reading of roles, but they do not exist in RTML. These constraints can be implemented by applications that use RTML and checked when credentials are being issued. Other constraints, like mutually exclusive permission are lost from the set reading.

Credentials:

K_{ABU}.university(name='StateU') $\longleftarrow$ K_{StateU} (1)

K_{StateU}.student(university='StateU', name='Bob Smith', program='M.S.', $\cdots$) $\longleftarrow$ K_{Bob} (2)

K_{ACM}.acmMember(number='UJ12345', name='Bob Smith', since=2000, $\cdots$) $\longleftarrow$ K_{Bob} (3)

Access rules of EPub

discount $\longleftarrow$ K_{ACM}.acmMember(name=?X, since$\leq$2001) $\cap$ student(name=?X, program$\in${'M.S.', 'Ph.D.'}) (4)

university() $\Longleftarrow$ K_{ABU} (5)

student(university=?X) $\Longleftarrow$ university(name=?X) (6)

From (1) and (5), EPub concludes that: university(name='StateU') $\longleftarrow K_{\text{StateU}}$ (7)

From (6) and (7), EPub concludes that: student(university='StateU') $\Longleftarrow K_{\text{StateU}}$ (8)

From (2) and (8), EPub concludes that: student(university='StateU', name='Bob Smith', $\cdots$) $\longleftarrow K_{\text{Bob}}$ (9)

From (3), (4), and (9), EPub concludes that: discount $\longleftarrow K_{\text{Bob}}$ (10)

Figure 1. Scenario 1: Linking Delegation and Intersection

StateU. It means that K_{StateU} is a member of the role "K_{ABU}.university(name='StateU')". Line (2) represents Bob's student credential issued by StateU. Line (3) represents Bob's ACM member credential.

Line (4) represents EPub's discount policy: Anyone who is both an ACM member since 2001 and a graduate student is entitled to the discount, and the name in the two credentials should be the same. Note that not all parameters appear in the role term for acmMember; only those that need to be constrained have to appear. The same is true for the role term for student. The RTML encoding of this policy in included in Appendix B.2.

Line (5) encodes EPub's delegation over the university role to K_{ABU}; this is called a *simple delegation*. The role term "university()" has no parameter at all, because the name parameter is not constrained. This delegation means that any principal that is a member of the role K_{ABU}.university(name=X) is also a member of EPub's university(name=X) role. In other words, it implies that the simple containment "university()$\longleftarrow K_{\text{ABU}}$.university()". When no restrictive inheritance is involved, these two are indeed equivalent. And the delegation syntax is simply a convenient syntactic sugar. In Section 3.3, we will illustrate their differences.

Line (6) encodes EPub's delegation over the identification of students of a university to principals who are certified to be that university. This is called a *linking delegation*. This implies that for any principal K and university name X, if K is a member of EPub's "university(name=X)" role, then EPub delegates the authority over the role "student(university=X)" to K.

3.2 Controlled Delegation of Permissions

Assume that a role, stanfordID, is declared in some ADSD; the details of the parameters of the stanfordID role are not important in this scenario. In another ADSD, a hostPerm role is declared and has one parameter: host. The type of the "host" parameter is "dns", which is pre-declared by the RTML system. (One can also use a type declared in the current ADSD via the type declaration mechanism provided by RTML.) In another ADSD, a socketPerm role is declared to *restrictively inherit* hostPerm, and a new parameter "port" is added, which has the pre-declared type "unsigned short".

Figure 2 gives the access rule, credentials, and some implications of them. Line (1) represents the delegation from FW to SA. The role K_{Stanford}.stanfordID() is called the *scope* of this delegation. Line (2) represents the delegation from SA to Alice. Line (3) represent Alice's Stanford ID credential. RTML encoding of the two ADSDs declaring hostPerm and socketPerm and credentials (2) and (3) are included in Appendix B.3.

When socketPerm is the only role that restrictively inherits hostPerm, then credential (1) is equivalent to the two credentials (4) and (5). The request, as represented by (6), is true because 'cs.stanford.edu' is a descendant of 'stanford.edu', and $8443 \in [8000..8443]$.

In general, when r' restrictively inherits r, then any definition "$K_A.r(\cdots) \longleftarrow e$" also implies "$K_A.r'(\cdots) \longleftarrow e$". Furthermore, any delegation "$K_A.r(\cdots) \Longleftarrow e$" also implies "$K_A.r'(\cdots) \Longleftarrow e$". The rationale is that when r' restrictively inherits r, then $r'(\cdots)$ represents a more restricted permission than $r(\cdots)$, and $r'(\cdots)$ is weaker than $r(\cdots)$ in the sense that any member of the $K_A.r(\cdots)$ role is also a member of the $K_A.r'(\cdots)$ role. This is achieved by having for each definition whose head uses r, also generating a definition that has head using r'.

Now we explain the difference between the delegation "$K_A.r() \Longleftarrow K_B$" and the containment "$K_A.r() \longleftarrow K_B.r()$". The delegation is stronger than the containment; it implies the containment (about r) and another containment about r': "$K_A.r'() \longleftarrow K_B.r'()$". The containment about r only implies "$K_A.r'() \longleftarrow K_B.r()$", which

Credentials:

K_{FW}.hostPerm(host $\in$ currentAndDescendants(‘cs.stanford.edu’)) $\Longleftarrow$ K_{SA} : K_{Stanford}.stanfordID() (1)

K_{SA}.socketPerm(host $\in$ descendants(’stanford.edu’), port$\in$ $[8000..8443]$) $\longleftarrow$ K_{Alice} (2)

K_{Stanford}.stanfordID($\cdots$) $\longleftarrow$ K_{Alice} (3)

Assuming socketPerm is the only role that restrictively inherits hostPerm, then (1) is equivalent to (4) and (5), in which we use t as a shorthand for “currentAndDescendants(‘cs.stanford.edu’)”:

K_{FW}.hostPerm(host $\in t$) $\longleftarrow$ K_{SA}.hostPerm(host $\in t$) $\cap$ K_{Stanford}.stanfordID() (4)

K_{FW}.socketPerm(host $\in t$) $\longleftarrow$ K_{SA}.sockerPerm(host $\in t$) $\cap$ K_{Stanford}.stanfordID() (5)

From (2), (3), and (5), the request, represented by the following query, should be authorized:

K_{FW}.socketPerm(host = ‘cs.stanford.edu’, port = 8443) $\longleftarrow$ K_{Alice} (6)

Figure 2. Scenario 2: Controlled Delegation of Permissions

is weaker than the containment about r' above, since any member of “$K_B.r()$” would also be a member of $K_B.r'()$.

3.3 Identity-based Separation of Duty

The bank creates one ADSD, which declares six roles: manager, cashier, auditor, twoCashiers, managerAndTwoCashiers, and approval. The latter three are declared to be manifold roles. For simplicity, we assume that these roles do not contain parameters.

Three definitions implementing the approval policy are given in Figure 3. Definition (1) means that FB’s twoCashiers role contains every principal set $\{K_1, K_2\}$ such that both K_1 and K_2 are members of FB’s cashier role, and $K_1 \neq K_2$. Definition (2) means that FB’s managerAndTwoCashiers role contains every principal set $p = \{K\} \cup p_1$ such that K is a member of FB’s manager role and p_1 is a member of FB’s twoCashiers role. Definition (3) means that the approval role contains every principal set $p = \{K\} \cup p_2$ such that K is a member of FB’s auditor role, p_2 is a member of FB’s managerAndTwoCashiers, and $K \notin p_2$.

Identity roles and identity-based roles

When an employee holds multiple keys, the definitions in Figure 3 may not achieve the goal of SoD, which requires different users rather than keys be responsible for the transaction. In decentralized and public-key based systems, one cannot assume that there is always an one-to-one relationships between keys and users. Such a relationship is often very difficult to enforce. In addition, there are often practical considerations that dictate one user having multiple keys. For example, a user may be required to change keys regularly, and to assure smooth transition, two keys may overlap. A user may also wish to have multiple keys in the interest of privacy and/or key security.

To address this problem, FB can declare a new role, employeeNumber, with one parameter, “number”, and declare this role to be an *identity role*. This means each physical user should correspond to one specific instance of the identity role. For example, if Carl’s employee ID with FB is ‘1111’, for any key K_D held by Carl, FB only issues “FB.employeeNumber(id=‘1111’) $\longleftarrow K_D$”, and no other employeeNumber credential containing K_D.

FB may then declare the roles twoCashiers, managerAndTwoCashiers, and approval to be based on the identity employeeNumber; we call these roles *identity-based*. In contrast, we call normal roles *principal-based*. Members of identity-based roles are computed based on the identities of principals.

We say that a principal K_D has an identity with respect to $K_A.r$ if for some parameters, denoted by “$\cdots$”, K_D is a member of $K_A.r(\cdots)$; and we say that $K_A.r(\cdots)$ is K_D’s identity wrt $K_A.r$. Two principals K and K' are *equivalent* wrt $K_A.r$, denoted by $K \equiv K'[K_A.r]$ (we omit “$[K_A.\mathrm{r}\,]$” when it is clear from the context), if K and K' are equal, or they have the same identity wrt $K_A.r$.

When using the rule “twoCashiers $\longleftarrow$cashier $\otimes$ cashier”, $\{K_1, K_2\}$ is a member of twoCashiers only when K_1 and K_2 each have identities and are not equivalent (wrt FB’s employeeNumber role), and each of K_1 and K_2 is equivalent to some member of FB’s cashier role, i.e., there exists two members K_1', K_2' of FB’s cashier role such that $K_1 \equiv K_1'$ and $K_2 \equiv K_2'$.

The notion we are using to ensure principals correspond to different users can be generalized to support applications of $\otimes$ to manifold roles. Two principal sets p_1 and p_2 are *equivalent* if every principal in p_1 is equivalent to one principal in p_2 and vice versa. Two principal sets p_1 and p_2 are *non-intersecting with respect to a given identity role* if all principals in p_1 and p_2 have identities and no principal in p_1 is equivalent to any principal in p_2. Thus, when using definition (3), the approval role contains every principal set $p = p_1 \cup p_2$ such that, p_1 and p_2 are non-intersecting, and there exists p_1' and p_2' such that $p_1 \equiv p_1'$, $p_2 \equiv p_2'$, p_1' is a member of FB’s auditor role (in which case p_1' must contain only one principal, since auditor is not a manifold role), and p_2' is a member of FB’s managerAndTwoCashiers role.

Access rules of FB

twoCashiers	$\longleftarrow$	cashier $\otimes$ cashier	(1)
managerAndTwoCashiers	$\longleftarrow$	manager $\odot$ twoCashiers	(2)
approval	$\longleftarrow$	auditor $\otimes$ managerAndTwoCashiers	(3)

Figure 3. Scenario 3: Identity-based Separation of Duty

We have seen that identity can be used in determining that two principals are not equivalent when a definition involves $\otimes$. Identity also affects other kinds of definitions. For example, when R is based on identity role r', and we have $K_A.R \longleftarrow K_A.R_1 \cap K_A.R_2$, if a user holds two keys K_1 and K_2 so that $K_1 \in K_A.R_1$ and $K_2 \in K_A.R_2$, then the user can get the permission encoded in $K_A.R$ when he can prove that $K_1 \equiv K_2$ wrt $K_A.r'$. In this case, we have $K_1 \in R$ and $K_2 \in R$. Again, this approach can be generalized to support applications of $\cap$ to manifold roles.

Our notion of identity role is somewhat similar to the notion of primary keys in databases. It is declared to uniquely identify something. This notion of identity is different from the traditional ones, in that it is not global. Not every user has to have an identity. Furthermore, different roles may be based on different identity roles, just as different relations in a databases may have different primary keys. We do not think that it is practical to have a globally unique identity for users at a global scale; however, it is often possible inside one organization.

Our approach does not solve the difficult problem of creating an infrastructure that uniquely identifies the holder of each principal. Rather we provide a mechanism to take advantage of such an infrastructure in policies when it is in place.

Extending inheritance and projection

One final issue bears on the Separation of Duty scenario. In practice, it is unlikely that a credential contains only an employee number. To avoid having to use multiple roles to document information about one employee (these multiple roles can still be included in one credential, as one credential can contain multiple definitions), one can do two things.

One approach is to first declare the employeeNumber role, then declare an employeeID role to *extend* employeeNumber and add additional parameters. We call this *extending inheritance*, by way of contrast with the notion of restrictive inheritance discussed in Section 3.2. In this case, membership in the employeeID role implies membership in the employeeNumber role. The intuition here is that everyone that has an employee ID has an employee number, plus some additional information carried in the new parameters. Thus, extending inheritance enables additional information to be added about role members, while restrictive inheritance enables additional requirements to be associated with a permission. When r' extends r, then for each principal K, $K.r(f_1 = ?X_1, \ldots, f_\ell = ?X_\ell)$ contains each member of $K.r'(f_1 = ?X_1, \ldots, f_\ell = ?X_\ell)$, in which $f_1, \ldots, f_\ell$ are all the parameters of r. Note that r' may contain additional parameters, but that the containment holds no matter what values they take.

Another approach is to first declare the employeeID role and then declare the employeeNumber role as a *projection* of the employeeID role to the number parameter. The effect is essentially the same as extending inheritance. Projection is useful when one wants to extract an identity role from a role that is already defined. Extending inheritance is useful, for instance, if one wants to update an old student ID, adding more parameters, but the new student ID credential should still prove membership in the old student ID role, since some applications may still use the old student ID role.

3.4 Summary of the Extensions

We now summarize the key new features added to RT, illustrated above by the scenarios. In Section 3.1, simple delegation and linking delegation are new. As used there, they are convenient syntactic sugars, simplifying the expression of requirements that can be equivalently expressed by using simple containment and linking containment (See Section 4.3). In Section 3.2, the tree types and the ability to use ranges in the head of a rule are new. Also new is the notion of restrictive inheritance, which creates a context in which simple delegation and linking delegation are not mere syntactic sugars, but capture otherwise inexpressible meanings. In Section 3.3, the notion of identity role, identity-based role, extending inheritance, and projection are new.

4 RTML

RTML is defined using XML Schema. The schema definition of RTML defines three top level elements: `Credential`, `AccessRule`, and `ApplicationDomainSpecification`. The schema for RTML uses data types in the XML Schema standard [2]; it also depends on the XML Signature standard [10], both for credential signatures and for representing public keys.

A `Credential` element can be divided into three parts: prologue, definitions, and verification data.

4.1 Prologue of a Credential

The prologue part of a credential has the following format.

```
<Preamble>
  <DefaultDomain uri="..." />
  <ImportDomain uri="..." name="..."> *
  <Principal> ...... </Principal> *
</Preamble>
<Issuer> ...... </Issuer>
<CredentialIdentifier> ......
</CredentialIdentifier>
```

The `Preamble` element contains reference information for the rest of the credential: a `DefaultDomain` element, zero or more `ImportDomain` elements, and zero or more `Principal` elements. The "uri" attribute of a `DefaultDomain` element specifies the location of an ADSD that acts as the default domain of the credential. Role names used in the credential are assumed to be declared in the default domain unless a domain is explicitly specified. An `ImportDomain` element has two attributes: a "uri" attribute that specifies the location of the ADSD to be imported, and a "name" attribute that is used to refer to the imported domain. Role names declared in imported domains can also be used in the current credential.

A `Principal` element gives the value of a principal, it can be a `KeyValue` element as defined in the XML Signature standard, an `IntegerValue` element, or a `StringValue` element. See Section 4.6 for more discussion of this. To improve readability, principals, which could be quite long, are included in the preamble so that they can be referred to in a compact way elsewhere in the credential.

The `Issuer` element contains a *principal value*, which may be a `Principal` element or a `PrincipalRef` element (which refers to a `Principal` element appearing in the preamble).

The `CredentialIdentifier` element contains a string that is unique among all credentials issued by the same issuer having the same default domain. It could be a serial number.

4.2 Roles in a Credential

After the prologue, a credential contains one or more definitions. Before introducing these definitions, we first explain the building blocks used in these definitions.

A *role* can take the form of a `RoleTerm` element (which is then assumed to be in the authority name space of the issuer) or an `ExternalRole` element (which contains a principal value and a `RoleTerm`).

A `RoleTerm` element has two attributes: name and domain (optional), which together identify a declared role. When the domain attribute is not present, the name attribute identifies a role declared in the default domain of this credential. When the domain attribute is present, it should be equal to the name attribute of one of the `ImportDomain` elements, and the name attribute identifies a role declared in the corresponding ADSD.

A `RoleTerm` element contains zero or more `Parameter` elements, each of which has two attributes: name (required) and id (optional). The id parameter uniquely identify the parameter in the current credential, so that it can be referred to elsewhere. A `Parameter` element optionally contains a constraint. A constraint may be a value of one of the seven categories of data types (which will be described in Section 4.6), a principal value, a `SpecialPrincipal`, an `Interval`, a `Set`, or an `Equals` element.

A constraint that is a value means that the parameter should be equal to this value. Currently, a `SpecialPrincipal` element can take one of two values: 'issuer' (which refers to the issuer of the current credential) and 'this'. The 'this' special principal can only be used when defining a singleton role; it refers to the principal being evaluated to be the member of the role. The following example from [20] illustrates the use of this: A company Alpha gives a pay raise to an employee if someone authorized to evaluate the employee says that his performance was good. This can be encoded using "payRaise $\longleftarrow$ evaluatorOf(this).goodPerformance".

An `Interval` element contains an optional `From` element and an optional `To` element, each of which contains a value of an ordered data type. This represents an interval set. The `From` (`To`) element has an attribute "included", indicating whether the bound in included in the interval. When the `From` (`To`) element is not present, that side of the interval is unbounded. A `Set` element includes one or more values. An `Equals` element has one attribute, which refers to another parameter in the definition, meaning that this parameter should equal the other parameter.

4.3 Definitions

There are eight kinds of definitions, each containing a `HeadRoleTerm` element and a body part. Before going into the definitions, we also need the notion of dimension. Each role has a *dimension*. A single-element role (default) has dimension 1. Manifold roles require that the dimension be explicitly declared. For a principal set to be a member of the role, its size must be no more than the role's dimension. For example, if R has dimension 2, then $\{K_1\}$ and $\{K_1, K_2\}$ may be members of $A.R$, but $\{K_1, K_2, K_3\}$ cannot be. The reason for requiring that the dimension be given is to ensure efficient evaluation. (See [20] for further details.)

Different kinds of definitions contain different elements as the body part. While describing these definitions, we use an abstract syntax, in which R represents the `HeadRoleTerm`, R_1 and R_2 represents other role terms, and Q (often with subscripts) represents roles, and in which we assume K_A is the issuer.

Simple Member $R \longleftarrow D$

The body part consists of one principal value, denoted D. This defines the principal D to be the member of the role $K_A.R$. More precisely, the role $K_A.R$ contains any principal that is equivalent to D. If R is principal-based, then equivalency is the same as equality. If R is based on an identity r', then the equivalency may also be determined by the identity.

Simple Containment $R \longleftarrow Q$

The body part consists of one role, denoted Q. The dimension of R should be no less than that of Q.

This defines the role $K_A.R$ to contain (every principal set that is equivalent to some principal set that is a member of) the role Q.

Intersection Containment $R \longleftarrow Q_1 \cap \cdots \cap Q_k$

The body part consists of an `Intersection` element, which contains two or more roles. The dimension of R should be no less than the maximum dimension of $Q_1 \cap \cdots \cap Q_k$.

This defines $K_A.R$ to contain the intersection of all the roles $Q_1, \ldots, Q_k$. More precisely, $K_A.R$ contains any principal set p that is equivalent to a member of Q_j, for every $j = 1..k$.

Linking Containment $R \longleftarrow R_1.R_2$

The body part consists of a `LinkedRole` element, which contains two `RoleTerm` elements. The dimension of R should be no less than that of R_2.

When R_1 is a singleton role, this defines the role $K_A.R$ to contain every $K_B.R_2$, in which K_B is a member of the role $K_A.R_1$. When R_1 is a manifold role, this defines, for any principal set $\{K_{B_1}, \ldots, k_{B_\ell}\}$ that is a member of $K_A.R_1$, $K_A.R$ contains the intersection of $K_{B_1}.R_2 \cap \cdots \cap K_{B_\ell}.R_2$.

Product Containment $R \longleftarrow Q_1 \odot \cdots \odot Q_k$

The body part consists of a `Product` element, which contains two or more roles. The dimension of R should be no less than the sum of the dimensions of $Q_1, \ldots, Q_k$.

This defines the role $K_A.R$ to contain every principal set p such that $p = p_1 \cup \cdots \cup p_k$ and for each $1 \leq j \leq k$, there exists p'_j such that $p_j \equiv p'_j$ and $p'_j \in Q_j$.

Exclusive Product Containment $R \longleftarrow Q_1 \otimes \cdots \otimes Q_k$

The body part consists of an `ExclusiveProduct` element, which contains two or more roles. The dimension of R should be no less than the sum of the dimensions of $Q_1, \ldots, Q_k$.

This defines the role $K_A.R$ to contain every principal set p that satisfies the following condition: $p = p_1 \cup \cdots \cup p_k$, for each $i \neq j$, $p_i \cap_{\equiv} p_j = \emptyset$ (p_i and p_j are non-intersecting), and for each $1 \leq j \leq k$, there exists p'_j such that $p_j \equiv p'_j$ and $p'_j \in Q_j$.

Simple Delegation $R \Longleftarrow B[: Q_c]$

The body part consists of a `DelegateTo` element (which contains a principal value) and a `Scope` element (which contains a role).

When Q_c is not present, K_A delegates its authority over R to K_B. In other words, K_A trusts K_B's judgement on assigning members to R. When Q_c is present, K_A wants to control its delegation such that K_B can only assign members of Q_c to be members of $K_A.R$, in other words, $K_A.R$ contains $K_B.R \cap Q_c$.

Linking Delegation $R \Longleftarrow R_1[: Q_c]$

The body part consists of a `DelegateTo` element (which contains a role name) and a `Control` element (which contains a role).

K_A delegates its authority over R to members of $K_A.R_1$. The delegation is restricted to members of Q_c. This implies $R \longleftarrow R_1.R \cap Q_c$.

4.4 Verification Data

The verification data part contains a `ValidityTime` element, zero or more `ValidityRule` elements, and one optional signature part.

The `ValidityTime` contains an `IssueTime` element, an optional `NotBefore` element, an optional `NotAfter` element, and an optional `LifeTime` element. The first three elements each contain a specific time (using the dateTime type in XML Schema), which we denote by t_i, t_b, and t_e. The `LifeTime` element contains a duration (using the duration type in XML Schema), we denote by δ.

A `ValidityRule` may specify a CRL location, an online verification server, etc. The details of the format of `ValidityRule` are still being worked out.

When a party receives a certificate, it must first check whether the validity period of this certificate has begun, i.e., whether the current time, t, is later than t_b. This allows post-dated credentials, such as a student ID that becomes valid only when the next academic year begins. The party next determines the fresh time t_f of the credential. Unless the authorizer checks the credential's validity, it assumes $t_f = t_i$. Validity rules define how checks can be performed to justify using a later fresh time. For example, when a CRL is checked and the credential is not revoked, one can update t_f to the issue time of the CRL (which is presumably later than t_i). The party then determines whether this certificate has expired. The expiration time is the earlier of t_e and $t_f + \delta$. Thus, by setting the life time δ, the issuer indicates that the credential should not be viewed as valid unless the party has checked for revocation sufficiently recently. Finally, the party determines whether the fresh time is sufficiently recent for its own purposes.

The optional signature part is a `Signature` element as specified in the XML Signature standard [10].

4.5 Access Rule

An `AccessRule` element is similar to a `Credential` element. The differences are as follows. An `AccessRule` does not have an `Issuer`; it has a `RuleIdentifier` instead of a `CredentialIdentifier`, it does not have `ValidityRule` or `Signature`. The rationale is that an access rule is created and used locally; as such, the issuer is implicit; no signature is needed; and a revoked rule is simply removed.

4.6 Application Domain Specification Documents (ADSD's)

An ADSD is represented by an "`ApplicationDomainSpecification`" element, which has a "uri" attribute, uniquely identifying this ADSD. An ADSD has the following structure.

```
<IncludeDomain uri="..."
         includeAll="true"|"false">
  <Type name="..."> *
  <RoleDeclaration name="..."> *
</IncludeDomain> *
<ImportDomain uri="..." name="..."> *
(type declaration) *
<PrincipalType>......</PrincipalType> ?
<RoleDeclaration ...> ...
</RoleDeclaration> *
```

Using Other ADSDs

In an `IncludeDomain` element, the "uri" attribute identifies the ADSD being included. When the "includeAll" attribute is true (default value), all the types and roles in the included domain are included in the current domain, i.e., they are considered to be as if declared in the current domain. When the "includeAll" attribute is false, one can select the types and roles being included by specifying them in the body of the element. One cannot include two types/roles that have the same name.

In an `ImportDomain` element, the "name" attribute serves as a short domain ID referring to the imported domain. One can use a type declared in an imported domain by using the domain ID together with the type name. Importing is useful when one wants to use two types that are declared in two ADSDs and have the same name.

RTML has a system domain, which declares some data types that are commonly used. Every ADSD automatically includes the system domain without using `IncludeDomain`, and so one can use these types freely.

Type Declarations

Every type declaration has a "name" attribute, which must take a value that is unique among all the data type names in a domain. One cannot declare a type if a type with the same name is already declared (possibly in an included domain). Types are organized into seven different categories; the details of type declarations are left to Appendix A because of space limitation.

Principal Type Declaration

By default, principals in RTML are public keys. However, one can override this by using a `PrincipalType` element to declare the principal type to be an integer type or a string type. This makes it possible to use RTML to encode policies that involve principals other than public keys and to use the RT system to make authorization decisions.

Whenever a role is declared, one needs to know which type of principal it contains. Every ADSD has at most one principal type. We say that an ADSD has a principal type if it includes a domain that already has a principal type, it contains a `PrincipalType` element, or it contains any role declarations at all. If an ADSD gets its principal type from declaring roles, the default public-key principal is assumed to be used. Roles that use different principal types must not be mixed in one credential.

The current RTML parser supports verification when principals are public keys. Credentials that are issued by principals of other types are not verified. Applications that use these credentials are responsible to perform any verification that is necessary.

Role Declarations

A `RoleDeclaration` element has five attributes: name, issuerTraces (default rule), subjectTraces (fact), dimension (1), and isIdentity (false). The "name" attribute should be unique among all role names in the current domain. The issuerTraces and subjectTraces attributes are related to distributed credential chain discovery, see [21] for more details. If the dimension is over 1, then the role is a manifold role. If the isIdentity attribute is set to true, then this is an identity role. When one issues credentials about the identity role, one should assign each physical user at most one unique combination of parameter values. Of course, each principal needs to determine which principal to trust about an identity role, just like every other role.

A `RoleDeclaration` element may optionally contain one of the following three elements: `Restriction` (for restrictive inheritance), `Extension` (for extending inheritance), and `Identity` (for identity-based roles).

Finally, a `RoleDeclaration` may contain zero or more `Parameter` elements; each one has a name attribute and contains a `Type` element.

5 Related Work

In this section, we compare RTML with X.509, SAML, SPKI/SDSI and KeyNote.

5.1 X.509

The basic authorization-related meaning of an X.509 certificate is easily mapped into RTML. In an X.509 certificate, the issuer attests to the association of a subject distinguished name (DN) and the subject key. If an X509 role is declared, then an X.509 certificate issued using key K_A to subject key K_B and subject DN $\{$C='US', O='StateU', OU='CSD', CN='Bob Smith'$\}$ can be represented as the simple member definition "K_A.X509(dn=$\{$C='US', O='StateU', OU='CSD', CN='Bob Smith'$\}$) $\longleftarrow K_B$". If the certificate also has CA capabilities, then it also represents the simple containment "K_A.X509(dn=?X) $\longleftarrow K_B$.X509(dn=?X)".

A X.509 certificate does not contain the issuer key; instead, it contains the DN of the issuer. A certificate is only useful when a certificate chain is obtained so that the issuer public key is determined. Given an ADSD encoding X.509 distinguished names, a standard X.509 certificate chain can be interpreted as a chain of RTML definitions, enabling existing certificates to be meaningful within an RTML system.

The above approach cannot capture additional purposes of an X.509 certificate beyond that of binding a DN to a key, nor purposes encoded in organization's non-standard extensions. However, an organization can enable others to translate implicit or explicit meanings of its X.509 certificates by providing an ADSD that describes the role information contained in them and a tool that extracts that role information from certificates.

5.2 SAML

The Security Assertion Markup Language (SAML) [13] is an XML-based framework for exchanging security information, expressed in the form of assertions about subjects. Assertions can convey information about authentication acts performed by subjects, attributes of subjects, and authorization decisions about whether subjects are allowed to access certain resources.

RTML has different purposes from SAML. SAML is used to convey results of authentication and authorization, but not credentials for doing so. RTML provides credential formats for documenting properties (expressed in the forms of roles) of subjects and documenting mechanisms to derive these properties such as delegation.

5.3 SPKI/SDSI and KeyNote

We compare RTML with SPKI/SDSI and KeyNote from three aspects: delegation structures, encoding of permissions, and support for separation of duty. The conclusions we draw are that RTML subsumes most of the expressive power of SPKI/SDSI and KeyNote and provides a lot of additional power. The places in which RTML is less expressive result from design trade-offs made in favor of guaranteeing properties such as the tractability of analyzing authorizations implied by RTML credentials and the interoperability and predictability that accrue to strong typing.

Delegation Structures

RTML has more expressive delegation structures than those in SPKI/SDSI and KeyNote.

SPKI has name certs and auth certs. Name certs can be represented in RTML using simple member, simple containment, and linked containment. An auth cert represents a delegation of the authority from its issuer to its subject, which can be a principal, a SDSI name, and a threshold structure. We will talk about threshold when discussing Separation of Duty later.

Assume that we are given an auth cert with issuer K_A, authority R, and subject K_B. If the delegation flag is false, then this is essentially "$K_A.R \longleftarrow K_B$". If the delegation flag is true, then this can be represented using two definitions: $K_A.R \longleftarrow K_B$ and $K_A.R \Longleftarrow K_B$. If one wants to use a SDSI name "K_B's N1's ... Nk" as a subject, one can define a new name "K_A's M" to have the same members[4] as "K_B's N1's ... Nk" and use $K_A.R \longleftarrow K_A.M$, and $K_A.R \Longleftarrow M$.

Note that since names in SDSI are only simple strings, one cannot represent "student(university=?X) $\Longleftarrow$ university(name=?X)" or "discount $\longleftarrow K_{\mathrm{ACM}}$.acmMember(since $\leq$ 2001)" in SPKI/SDSI. Furthermore, SPKI/SDSI does not have intersection, which is needed in Scenario 1, or controlled delegation, which is needed in Scenario 2. When a principal allows a subject to further delegate a permission, it cannot restrict to whom the subject delegates.

A KeyNote *request* is characterized by a list of fields, which are name/value pairs. In KeyNote, credentials and policies (access rules) are called assertions. An assertion has *conditions* written in an expression language, which

[4]This can be achieved by introducing new intermediate roles and definitions. See [21] for details.

refers to fields in requests. The intuitive meaning of an assertion is that, if the licensees support a request, and the request satisfies the conditions, then the issuer supports the request as well.

KeyNote's delegation structures in assertions are more restricted than those in SPKI/SDSI. Delegation in KeyNote is always transitive, one cannot grant a permission to a principal without enabling the principal to further grant the permission. Furthermore, KeyNote assertions have to explicitly list the principals involved in the delegation. Therefore, using KeyNote assertions, one cannot express a delegation "student() $\Longleftarrow$ university()" (which is expressible in SPKI/SDSI). See [18, 20] for more discussion of this limitation.

Encoding of Permissions

Permissions are encoded in RTML by using role terms, whose parameters are typed and can be constrained.

In SPKI, authorities are encoded in tags, which are untyped lists, e.g., (ftp (ftp.stanford.edu) (* prefix /pub/test/)). One cannot express the permission to connect all hosts in a domain, since the root of a domain goes at the end of the string, and SPKI does not have a (* suffix) operator. There are also other limitations. For example, one cannot encode a permission that requires two parameters being equal.

While it is not clear that everything that can be expressed in tags can be expressed using role terms, we do claim the following. First, all the examples we encountered in SPKI literature can be expressed using role terms. Second, the ability to flexibly refine permissions that is allowed by untyped lists in SPKI can be achieved by using restrictive inheritance in RTML. Third, the untyped list approach in SPKI has been found to have certain problems. For example, in [15], it has been shown that the intersection between two tags may not be finitely representable using tags.

In KeyNote, the permissions delegated in an assertion are represented by conditions on fields. These conditions are very expressive, including formula constructed using integers with function symbols $\{+, -, *, /, \%, \hat{}\}$ and predicates $\{=, \neq, <, >, \leq, \geq\}$. KeyNote conditions also include regular expressions. We believe that this is more expressive than role terms in RTML. However, this expressiveness of KeyNote comes at the cost of the ability to analyze KeyNote assertions. In [19], it has been shown that it is undecidable to compute the set of all requests that a set of KeyNote assertions authorizes. Note that whether any specific request is authorized by a set of assertions can still be determined efficiently. However, there does not exist an algorithm to perform analysis of all the requests being authorized by a set of assertions. In fact, it is undecidable even when there is only one assertion delegating to a single entity, and the question is just whether the assertion authorizes any request at all. On the other hand, the constraints in RTML are designed so that the implications of a set of RTML credentials can be efficiently computed. We feel that the constraints in RTML provides sufficient expressive power for most applications, as all the examples in [3] can be expressed in RTML.

Support for Separation of Duty

Both SPKI and KeyNote allow delegation to k-out-of-n threshold structures, in which one explicitly lists the n principals. It has been argued before that such threshold structures are inconvenient [17]. For example, to express the access rule "twoCashier $\longleftarrow$cashier $\otimes$ cashier", one needs to explicitly list all the cashiers in the access rule, and this rule needs to be changed each time members in the cashier role change.

A threshold structure requires agreement of multiple principals drawn from a single list. When the policy is to require different principals drawn from the membership of different roles, it is not clear that threshold can help. It seems that the policy writer needs to enumerate all the principal sets that are entitled to the permission and to delegate to each of them directly.

When a user may hold more than one keys, thresholds seem to be useless. For example, if n users each hold 2 keys, then a policy that requires 2 out of n users would require 2^n different thresholds using the straightforward approach. It is better to directly express all the pairs that are eligible to access, as there are $2n(n-1)$ such pairs.

6 Conclusion

This paper describes the following advances in the RT framework that broaden its applicability: new data types to encode permissions involving structured resources and ranges; restrictive inheritance of roles for flexible refinement of permissions; and notions of identity roles and identity-based roles to address issues when a physical user holds multiple keys. In addition to these extensions motivated by specific applications, this paper describes RTML, an XML-based data representation for RT policies and credentials. RTML establishes a precise format for RT credentials and policies, help enabling the deployment of the RT framework.

Compared with previous TM systems such as SPKI/SDSI and KeyNote, RTML has the following distinguishing features.

- RTML supports more flexible delegation. In RTML, one can delegate to principals who are members of certain roles, and can control the scope of a delegation.

- RTML addresses the issue of vocabulary agreement. Application Domain Specification Documents in RTML ensure uniqueness of role names, and enable credentials to be strongly-typed, further helping to ensure interoperability and to reduce the possibility of errors in writing policies and credentials and unintended interaction of credentials.

- RTML supports Separation of Duty policies in a way that is more expressive than previous TM systems. Furthermore, RTML addresses the situation when one user holds more than one keys.

References

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

[2] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. W3C Recommendation, May 2001.

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

[4] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer, 1999.

[5] 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.

[6] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance-checking in the PolicyMaker trust management system. In *Proceedings of Second International Conference on Financial Cryptography (FC'98)*, volume 1465 of *Lecture Notes in Computer Science*, pages 254–274. Springer, 1998.

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

[8] 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.

[9] 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.

[10] Donald Eastlake, Joseph Reagle, and David Solo. XML-Signature Syntax and Processing. W3C Recommendation, February 2002.

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

[12] David C. Fallside. XML Schema Part 0: Primer. W3C Recommendation, May 2001.

[13] Phillip Hallam-Baker and Eve Maler. Assertions and protocol for the oasis security assertion markup language (saml). OASIS Committee Specification, May 2002.

[14] Russell Housley, Warwick Ford, Tim Polk, and David Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. IETF RFC 2459, January 1999.

[15] Jonathan R. Howell. *Naming and sharing resources acroos administrative boundaries*. PhD thesis, Dartmouth College, May 2000.

[16] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[17] 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.

[18] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, 6(1):128–171, February 2003.

[19] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, pages 58–73. Springer, January 2003.

[20] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[21] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, February 2003.

[22] Ronald L. Rivest and Bulter Lampson. SDSI — a simple distributed security infrastructure, October 1996. Available at http://theory.lcs.mit.edu/∼rivest/sdsi11.html.

[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] Kent E. Seamons, Marianne Winslett, and Ting Yu. Limiting the disclosure of access control policies during automated trust negotiation. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'01)*, February 2001.

[25] Kent E. Seamons, Marianne Winslett, Ting Yu, Bryan Smith, Evan Child, Jared Jacobsen, Hyrum Mills, and Lina Yu. Requirements for policy languages for trust negotiation. In *Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks (Policy 2002)*, pages 68–79. IEEE Computer Society Press, June 2002.

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

[27] William H. Winsborough and Ninghui Li. Towards practical automated trust negotiation. In *Proceedings of the Third International Workshop on Policies for Distributed Systems and Networks (Policy 2002)*, pages 92–103. IEEE Computer Society Press, June 2002.

[28] William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, volume I, pages 88–102. IEEE Press, January 2000.

[29] Ting Yu, Marianne Winslett, and Kent E. Seamons. Interoperable strategies in automated trust negotiation. In *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS-8)*, pages 146–155. ACM Press, November 2001.

A Type Declarations in ADSDs

In the following, we present type declarations in ADSDs. Every type declaration has a “name” attribute, which we omit in the presentation below. Under each category, we also describe how a constant value of a type in that category is represented.

Integer types

An `IntegerType` element has two additional required attributes: max and min. It also has four optional attributes: step (default 1), base (0), includeMin (true), and includeMax (true). The legal values of this type include all integer value v’s such that $v = \mathrm{base} + k * \mathrm{step}$ for some integer k and that $min \leq v \leq max$, where $\leq$ ($\geq$) should be replaced by $<$($>$) if includeMin (includeMax) is set to false. For example, one can declare a type to contain all the numbers v such that $v \mod 3 = 1$ and $0 \leq v \leq 100$.

A constant of an integer type is represented using an `IntegerValue` element. The following integer types are declared in the system domain: long, int, short, byte, bit, unsigned int, unsigned short, and unsigned byte.

Decimal types

A `DecimalType` element is similar to an `IntegerType` element, in that it has the same attributes. However, the attributes min, max, base, and step now take decimal values. Valid values of this type are defined in the same way as are those of an integer type. A constant is represented using a `DecimalValue` element. No decimal type is declared in the system domain.

Enumeration Types

An `EnumType` element has three optional attributes: ignoreCase (false), which specifies whether to ignore case when comparing two enumeration values; ordered (false), which specifies whether this type is ordered; and size, which specifies how many values this type has. If this type is ordered, then one can use intervals to constrain parameters of this type. The `EnumType` element contains a list of `EnumValue` elements, which enumerates the legal values of this type.

A constant is represented using an `EnumValue` element. The system domain contains a boolean type, which is declared as an EnumType. Other possible examples of enumeration types include day of week, degree, etc.

String types

A `StringType` element has two optional attributes: ignoreCase (false) and ordered (false), which have the same meanings as in the case of enumeration types. A constant is represented using a `StringValue` element. The system domain declares two types of this category: string and case-insensitive string.

Tree types

A `TreeType` element has two additional required attributes: separator and order. These two values determine what a tree value looks like. For example, a type for DNS names has “.” as separator and the order is “rootLast”, while a type for Unix file paths has “/” as separator and the order is “rootFirst”.

A `TreeValue` element contains a string such as “/usr/home” and three optional attributes: includeCurrent (default true), includeChildren (false), and includeDescendants (false). The default value means that only the current node is included. One can set these attributes to reflect other choices. Note that children are considered to be a subset of descendants, and so when includeDescendants is set to true, all children are also included, no matter what the value of includeChildren is.

The system declares two types of this category: dns and path.

Record types

A `RecordType` element contains one or more `Field` elements; each has a name attribute and contains a `Type` element. The `Type` element has two attributes: name (required) and domain (optional); they refer to a type already declared, i.e., declared in an included domain or before the current declaration. This guarantees that no recursion occurs with record types.

A `Record` element can be used to constrain a parameter of a record type; it contains one or more fields, each having an optional constraint.

Examples of record types include IP addresses, names, and street addresses.

Date/Time types

Date/Time types are treated differently from other data types. RTML borrows the following standard date/time types defined in XML Schema [2]: date, time, dateTime, gYear, gYearMonth, gMonth, gMonthDay, and gDay. These types can be used as if they are declared in the system domain. RTML does not support defining new data/time types. A `TimeValue` element can contain any value that is legal for one of the above types .

B Sample ADSDs and Credentials in XML

B.1 The System ADSD

The system ADSD consists of the data types declared as follows and date/times from XML Schema. It is automatically included in every other ADSD.

```
<?xml version="1.0" encoding="UTF-8"?>
<ApplicationDomainSpecification uri=""
    xmlns="http://crypto.stanford.edu/dc/RTMLv1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://crypto.stanford.edu/dc/RTMLv1.0
                        http://crypto.stanford.edu/˜ninghui/rtml/RTMLv1.0q.xsd">
  <IntegerType name="long" max="9223372036854775807" min="-9223372036854775808"/>
  <IntegerType name="int" max="2147483647" min="-2147483648"/>
  <IntegerType name="short" max="32767" min="-32768"/>
  <IntegerType name="byte" max="127" min="-128"/>
  <IntegerType name="bit" max="1" min="0"/>
  <IntegerType name="unsigned int" max="4294967295" min="0"/>
  <IntegerType name="unsigned short" max="65535" min="0"/>
  <IntegerType name="unsigned byte" max="255" min="0"/>
  <EnumType name="boolean">
    <EnumValue>false</EnumValue>
    <EnumValue>true</EnumValue>
  </EnumType>
  <StringType name="string"/>
  <StringType name="case-insensitive string" ignoreCase="true"/>
  <TreeType name="dns" separator="." order="rootLast"/>
  <TreeType name="path" separator="/" order="rootFirst"/>
  <RecordType name="email address">
    <Field name="user name"> <Type name="string"/> </Field>
    <Field name="server"> <Type name="dns"/> </Field>
  </RecordType>
  <RecordType name="person name">
    <Field name="first name"> <Type name="string"/> </Field>
    <Field name="last name"> <Type name="string"/> </Field>
  </RecordType>
  <RecordType name="distinguished name">
    <Field name="CN"> <Type name="string"/> </Field>
    <Field name="OU"> <Type name="string"/> </Field>
    <Field name="O"> <Type name="string"/> </Field>
    <Field name="CN"> <Type name="string"/> </Field>
  </RecordType>
</ApplicationDomainSpecification>
```

B.2 Sample XML Elements for Scenario 1

The following definition corresponding to EPub’s discount policy in Scenario 1.

```
<IntersectionContainment>
  <HeadRoleTerm name="Discount"/>
  <Intersection>
    <ExternalRole>
      <PrincipalRef ref="K_ACM"/>
      <RoleTerm name="ACM Member">
        <Parameter name="name" id="memberName"/>
        <Parameter name="since">
```

```
          <Interval>
            <To><TimeValue>2001</TimeValue></To>
          </Interval>
        </Parameter>
      </RoleTerm>
    </ExternalRole>
    <RoleTerm name="student">
      <Parameter name="name"> <Equals ref="memberName"/> </Parameter>
      <Parameter name="program">
        <Set>
          <EnumValue>M.S.</EnumValue>
          <EnumValue>Ph.D.</EnumValue>
        </Set>
      </Parameter>
    </RoleTerm>
  </Intersection>
</IntersectionContainment>
```

B.3 Sample XML Elements for Scenario 2

The following is the ADSD that declares the “hostPerm” role.

```
<ApplicationDomainSpecification
    uri="http://crypto.stanford.edu/˜ninghui/rtml/examples/FWADSD1.xml"
    xmlns="http://crypto.stanford.edu/dc/RTMLv1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://crypto.stanford.edu/dc/RTMLv1.0
                        http://crypto.stanford.edu/˜ninghui/rtml/RTMLv1.0q.xsd">
  <RoleDeclaration name="hostPerm">
    <Parameter name="host"> <Type name="dns"/> </Parameter>
  </RoleDeclaration>
</ApplicationDomainSpecification>
```

The following is the ADSD that declares the “socketPerm” role, which restrictively inherits “hostPerm”.

```
<ApplicationDomainSpecification
    uri="http://crypto.stanford.edu/˜ninghui/rtml/examples/FWADSD2.xml"
    xmlns="http://crypto.stanford.edu/dc/RTMLv1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://crypto.stanford.edu/dc/RTMLv1.0
                        http://crypto.stanford.edu/˜ninghui/rtml/RTMLv1.0q.xsd">
  <IncludeDomain uri="http://crypto.stanford.edu/˜ninghui/rtml/examples/FWADSD1.xml"/>
  <RoleDeclaration name="socketPerm">
    <Restriction> <BaseRole name="hostPerm"/> </Restriction>
    <Parameter name="port"> <Type name="unsigned short"/> </Parameter>
  </RoleDeclaration>
</ApplicationDomainSpecification>
```

The following is the definition element of K_{FW}’s delegation to K_{SA}.

```
<SimpleDelegation>
  <HeadRoleTerm name="hostPerm">
    <Parameter name="host">
      <TreeValue includeCurrent="true" includeDescendents="true">
        cs.stanford.edu
      </TreeValue>
    </Parameter>
  </HeadRoleTerm>
  <DelegateTo>
    <PrincipalRef ref="K_SA"/>
  </DelegateTo>
  <Control>
    <ExternalRole>
      <PrincipalRef ref="K_Stanford"/>
      <RoleTerm name="StudentID" domain="Stanford"/>
    </ExternalRole>
  </Control>
</SimpleDelegation>
```

The following is the definition element of K_{SA}'s delegation to K_{Alice}.

```
<SimpleMember>
  <HeadRoleTerm name="socketPerm">
    <Parameter name="host">
      <TreeValue includeDescendents="true" includeCurrent="false">stanford.edu</TreeValue>
    </Parameter>
    <Parameter name="port">
      <Interval>
        <From><IntegerValue>8000</IntegerValue></From>
        <To><IntegerValue>8443</IntegerValue></To>
      </Interval>
    </Parameter>
  </HeadRoleTerm>
  <PrincipalRef ref="K_Alice"/>
</SimpleMember>
```

B.4 A Complete, Signed Credential

We now give a complete credential. The signature is generated using the Apache XML Security tool.

```
<Credential
        xmlns="http://crypto.stanford.edu/dc/RTMLv1.0"
        xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://crypto.stanford.edu/dc/RTMLv1.0
                            http://crypto.stanford.edu/~ninghui/rtml/RTMLv1.0q.xsd
                            http://www.w3.org/2000/09/xmldsig#
                            http://www.w3.org/TR/xmldsig-core/xmldsig-core-schema.xsd">
  <Preamble>
    <DefaultDomain uri="http://crypto.stanford.edu/dc/rtml/x509adsd.xml">
    </DefaultDomain>
    <Principal id="IssuerKey">
      <ds:KeyValue>
        <ds:DSAKeyValue>
          <P>
            /X9TgR11EilS30qcLuzk5/YRt1I870QAwx4/gLZRJmlFXUAiUftZPY1Y+r/F9bow9subVWzXgTuA
            HTRv8mZgt2uZUKWkn5/oBHsQIsJPu6nX/rfGG/g7V+fGqKYVDwT7g/bTxR7DAjVUE1oWkTL2dfOu
            K2HXKu/yIgMZndFIAcc=
          </P>
          <Q>l2BQjxUjC8yykrmCouuEC/BYHPU=</Q>
          <G>
            9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6EwoFhO3
            zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfBpKL
            Zl6Ae1UlZAFMO/7PSSo=
          </G>
          <Y>
            vLpQw3oYKp/iAL8drwI1teVtu5TGt8+1Z7YyUuI/ztvd0ittFVw/udC7HEyLF1A34saKGoES3X3V
            wsr9ilpx6e1tHFSHHVo87GsDXdNlIKUKkJhtysttrlOStBG7hcKcdVISdaw/Pvyfod5oAhTA0Tw1
            9sAeigAelUO4qsyr/20=
          </Y>
        </ds:DSAKeyValue>
      </ds:KeyValue>
    </Principal>
    <Principal id="SubjectKey">
      <ds:KeyValue>
        <ds:RSAKeyValue>
          <ds:Modulus>
            ujN6AfAP1GhzwiXlP2DwJod5ivWw7bnQA903bTQmMhN1kkPxcSEmMPW1f+yof3cza0Xz9WgeBc9+
            XwM15Ot/J4KGYHoDrLlyr1A2uKnRtixJphpJGCbw09CoCAHEwC25+93c7aG1j3kWoKBQqn9fCH3s
            QO5dt1DxNoq3ah0jq0c=
          </ds:Modulus>
          <ds:Exponent>AQAB</ds:Exponent>
        </ds:RSAKeyValue>
      </ds:KeyValue>
    </Principal>
  </Preamble>
  <Issuer><PrincipalRef ref="IssuerKey"></PrincipalRef></Issuer>
  <SimpleMember>
    <HeadRoleTerm name="DistinguishedName">
      <Parameter name="subjectDN">
        <Record>
          <Field name="CN"><StringValue>Bob Smith</StringValue></Field>
          <Field name="OU"><StringValue>CSD</StringValue></Field>
          <Field name="O"><StringValue>StateU</StringValue></Field>
          <Field name="C"><StringValue>US</StringValue></Field>
        </Record>
      </Parameter>
    </HeadRoleTerm>
    <PrincipalRef ref="SubjectKey"></PrincipalRef>
  </SimpleMember>
  <ValidityTime>
    <IssueTime>2002-08-20T13:20:00Z</IssueTime>
    <NotAfter>2003-08-20T13:20:00Z</NotAfter>
  </ValidityTime>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
      </CanonicalizationMethod>
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1">
      </SignatureMethod>
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature">
          </Transform>
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments">
          </Transform>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
        <DigestValue>vgeAJujgY/eHjj0ReTKAywqSPk8=</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>cCH3MJMCY1Bb1MGn5HYfS4mHrApVhguBNFjAHjV+5MuCLdemhyC61Q==</SignatureValue>
    <KeyInfo>
      <KeyValue>
```

```
        <DSAKeyValue>
          <P>
            /X9TgR11EilS30qcLuzk5/YRt1I870QAwx4/gLZRJmlFXUAiUftZPY1Y+r/F9bow9subVWzXgTuA
            HTRv8mZgt2uZUKWkn5/oBHsQIsJPu6nX/rfGG/g7V+fGqKYVDwT7g/bTxR7DAjVUE1oWkTL2dfOu
            K2HXKu/yIgMZndFIAcc=
          </P>
          <Q>l2BQjxUjC8yykrmCouuEC/BYHPU=</Q>
          <G>
            9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6EwoFhO3
            zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfBpKL
            Zl6Ae1UlZAFMO/7PSSo=
          </G>
          <Y>
            Eln5/htZP51p7Y/Y1+zZOSSmoi2fQS0deniScan3990xy33RrPfF5odqEVmVYfTzFfKEz94aUXEY
            qY2VGVRCKrAZThk1SwoOB+UyfNSVjoqa4fppIQpTalK/JeR7uxQUr0Aeop68nr2u49GijYiLyvL3
            x04lGaZ8jUYZL3gZTNI=
          </Y>
        </DSAKeyValue>
      </KeyValue>
    </KeyInfo>
  </Signature>
</Credential>
```