Generalized and Practical Role-Based Cascaded Delegation

Danfeng Yao, Roberto Tamassia, Seth Proctor

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-05-01
March 2005

Generalized and Practical Role-Based Cascaded Delegation*

Danfeng Yao† Roberto Tamassia† Seth Proctor‡

Abstract

A decentralized trust management system needs to be efficient and scalable to keep low communication and computation costs. Because in distributed and pervasive computing devices may have small storage units and limited bandwidth, lengthy delegation credentials are inefficient to store or transmit. We discuss how to improve the efficiency and flexibility of role-based delegations in ubiquitous computing environment. A set of operations for managing delegation chains in role-based cascaded delegation (RBCD) model are presented. These operations provide significantly improved efficiency and flexibility for the RBCD model, without increasing management complexities.

Keywords: Cascaded delegation, Decentralized trust management, Role-based access control

1 Introduction

Trust management systems provide access control in environments where initially unknown entities from different administrative domains interact and establish trust through mutually trusted parties. Several such systems have been proposed, e.g., KeyNote [7], SPKI/SDSI [11], OASIS [2], *RT* framework [15], and the RBCD protocol [21]. Role-based decentralized delegation [15, 21] combines trust management with role-based access control (RBAC) [19]. In such models, privileges can be delegated to roles, and therefore delegation is efficient and scalable. Role-based cascaded delegation (RBCD) [21] is a delegation model that facilitates large-scale dynamic resource sharing in decentralized, pervasive collaborative environments, and is suitable for tasks where coalitions are dynamically formed by role members from different administrative domains.

The RBCD protocol comprises four operations: INITIATE, EXTEND, PROVE, and VERIFY [21]. INITIATE and EXTEND operations are used by a resource owner and an intermediate delegator, respectively, to delegate a privilege to a role. PROVE operation is used by a requester to produce a proof of a delegation chain that connects the resource owner with the requester. VERIFY operation decides whether the requester is granted access based on the proof. Delegation credentials in RBCD are accumulated at each delegation transaction and form a delegation chain. A delegation chain represents how trust or a delegated privilege is transferred from one role to another. It contains a sequence of delegation credentials that connects unknown entities and resource owners. For example, a user with role *manager* at Central Bank issues a delegation credential C_1 to role *clerk* at a state bank to authorize the right of accessing a document *doc*. A member of role *clerk* at the state bank delegates this right to a role at a county bank by issuing another delegation credential C_2. Credential C_1 and C_2 form a delegation chain connecting the role at the county bank with the Central Bank. An efficient RBCD protocol [21] with compact delegation credentials can be realized based on the hierarchical certificate-based encryption scheme [13].

*Work supported in part by a research gift from Sun Microsystems, Inc. and by NSF grant IIS–0324846.

†Computer Science Department, Brown University, Providence, RI, 02912, {dyao, rt}@cs.brown.edu

‡Sun Microsystems Laboratories, Burlington, MA, Seth.Proctor@sun.com

In this paper, we generalize the original RBCD model to improve its security, efficiency, and flexibility. Next three paragraphs capture the motivations for these improvements. In the RBCD model, a valid role member who is delegated a privilege can further extend that privilege to others, without the participation of role administrators. However, giving delegation powers to individuals crucially relies on the trustworthiness of role members. Therefore, we need to improve the expressive power of the RBCD delegation language, so that resource owners and intermediate delegators can restrict delegation scopes and issue fine-grained delegation credentials.

A decentralized trust management system should be efficient in dynamic access control environments such as an emergency room (ER) scenario. Suppose that during an operation on a patient, doctors find that the help of experts from several other organizations are urgently needed. These experts need to be given temporary access to medical information related to the patient, in order for them to make medical decisions. Issuing authorizations through regular administration channels may not be as efficient as needed. The existing RBCD model is particularly suitable for this dynamic delegation scenario, because doctors can issue delegations based on their credentials without administrator's participation. However, in RBCD if a doctor wants to delegate privileges associated with several credentials (e.g., doctor role credential, ER access credential and medical consortium membership credential), he has to generate one delegation certificate corresponding to each of these credentials. Furthermore, the doctor has to repeat this delegation process for each of the roles that receive the delegations. Therefore, the delegation efficiency in RBCD needs to be further improved for better performance in dynamic and emergency access control situations.

The role-based credential accumulation approach of RBCD avoids the need for dynamic credential chain discovery, which may be potentially costly in terms of communication overhead. However, the ability to dynamically construct delegation chains is important for a flexible delegation model. This prompts us to investigate the possibility of integrating credential chain discovery with the RBCD model. There are several security and performance considerations for such an integrated model. Because delegation chains constructed by discovery algorithms may give authorizations to a large number of unknown users, a resource owner should be able to control the availability of its delegations to the credential discovery. To achieve this, existing credential chain discovery algorithms [16] need to be modified, so that credential issuers can specify how their privileges are propagated through credential chain discovery. In addition, communication costs need to be considered in credential chain discovery for better performance in distributed settings. We extend the RBCD model [21] to include the support of credential chain discovery algorithms. The contributions of this paper are summarized next.

1.1 Our contributions

In this paper, we make several practical extensions to the original RBCD model [21] and we present an RBCD implementation in Java and C++. Our contributions are summarized as follows.

1. We enrich the expressiveness of the RBCD language model to support the use of delegation predicates and constraints. Using these specifications the scope of delegation recipients can be flexibly adjusted. At VERIFY operation, predicates are evaluated with attribute values obtained from requester's credentials. In Section 2.1, we describe a language model that supports predicates and constraints. Examples of using them are given in Section 3, and how this addition changes the RBCD protocol operations is described in Section 4.1.
2. We describe a new operation of the RBCD protocol that gives intermediate delegators the ability to largely reduce the number of delegation credentials needed be generated. In the original RBCD model [21], if an intermediate delegator E wants to extend n delegation credentials to m roles, then E has to run the EXTEND operation $n \times m$ times to compute delegation credentials. If n or m is large, this may not be efficient in terms of running time, storage space, and transmission speed. To improve the delegation efficiency, we propose a new operation MERGE EXTEND that first extends each of the n delegation credentials to a new role r, and then further delegates role r to the m recipient roles. MERGE EXTEND operation reduces the time

required for generating delegation credentials, as an intermediate delegator only needs to perform n EXTEND operations and m INITIATE operations. Because role r is locally defined by an intermediate delegator E and is unknown to resource owners, entities who are delegated role r need to show the proofs of privileges associated with r. We describe how this is achieved in our cascaded delegation model. Merged credentials can be easily *split* back into individual credential chains. This flexibility of manipulating credentials allows an intermediate delegator to extend multiple delegation chains or a portion of merged privileges.

3. We show how to improve the flexibility of trust management in the RBCD model, by dynamically constructing longer delegation chains from RBCD credentials distributed across the network. We call this a combined approach, because it integrates distributed credential chain discovery algorithm into the RBCD protocol. This model has many advantages. The use of RBCD credentials partially reduces communication and computation costs, because portions of delegation chain are already captured in RBCD delegation credentials. Credential chain discovery improves the delegation flexibility, because credential holders can establish trust with a larger number of resources through the discovery. To achieve a tighter security and better performance, we also modify the existing credential chain discovery algorithm and use it in our combined model. We show that delegation chains returned by discovery algorithms may give authorizations to more unknown users than in RBCD models, because there is a semantic difference between a discovered delegation chain and a delegation chain generated by EXTEND operations of RBCD. Therefore, we give resource owners the ability to distinguish these two cases, by using attribute constraints.
4. Finally, we implement a prototype of RBCD in Java and C++, which uses the package java.security, BLS short signature [10] implementation in C++ [3, 4], and the Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL) [18]. We describe the layered architecture of our implementation. Lower layers have BLS signature implementation in C++, and programs that wrap C++ signature operations in Java. Upper layers are entirely written in Java, and contain classes for basic certificates, keys, and signatures. Classes in java.security and java.security.cert are extended for compatibility considerations with other Java applications. Classes for RBCD credential and operations constitute the top layer of our architecture. Our prototype implements INITIATE, EXTEND, PROVE, and VERIFY operations. We present and analyze our experimental results on the running time of RBCD operations and sizes of RBCD delegation credentials. INITIATE, EXTEND, and PROVE operations are quite fast and have an average running time about 8 ms. In comparison, VERIFY operation is more time-consuming due to the long verification time of BLS signatures. The use of short signatures [10] effectively reduces the size of credentials in RBCD, compared to RSA signatures.

1.2 Organization of the paper

The rest of this paper is organized as follows. Definitions and our language model are given in Section 2. In Section 3, an example illustrates merging RBCD credentials in dynamic collaboration in an emergency medical situation. Then, the *merge* RBCD (mRBCD) protocol is presented in Section 4. We describe how credential chain discovery is combined with RBCD in Section 5. Our Java and C/C++ implementation of the original RBCD protocol is presented in Section 6. Section 7 analyzes our experimental results and addresses issues of security and efficiency in our models. The related work is given in Section 8. Finally, Section 9 contains conclusions and future work.

2 Terminology and language model

As in the original RBCD model [21], we define an *entity* to be either an organization or an individual. An entity may issue credentials and make requests. Also, an entity may have one or more affiliated roles or delegated roles, which are authenticated by credentials. An *affiliated role credential* is the credential for an affiliated role, and is signed by the administrator of the role. A *delegation credential* is the credential for proving a delegated privilege. A *privilege* can be role assignment(s) or action(s) on a resource. An *extension credential* or *extension certificate* is generated and signed by a delegator on delegation transaction information, such as identities of the delegator and delegatee, and the delegated privilege. A *complete delegation credential* includes a signature from a requester, extension credentials, and role credentials. It authenticates a delegation chain of a privilege connecting a resource owner with a requester. That is, the original issuer is the resource owner, intermediate delegators are members of the immediately previous recipient roles, and the requester is a member of the last recipient role. A *partial delegation credential* is a delegation credential issued to a role. It cannot be used by an individual for proving authorization, as it lacks the identity and role information of the requester. In this paper, we use curly brackets (for example $\{C\}$) to refer to a set of RBCD delegation credentials.

2.1 Language model

Role r administered by entity A is denoted as $A.r$, as in the original RBCD model [21]. Entity A is the administrator of role $A.r$. A role defines a group of entities who are members of this role. If an entity D has an affiliated role $A.r$, her *role credential* is denoted by $A \xrightarrow{A.r} D$, which indicates that D is assigned role $A.r$ by the role administrator A. Entity D can delegate role $A.r$ to a role $B.s$ (administered by B) by issuing an *extension credential*, which is denoted by $D \xrightarrow{A.r} B.s$. The above delegation means that entity A delegates to role $B.s$ privileges that are associated with role $A.r$ *and* are controlled by entity A. It does not automatically grant $B.s$ the privileges that are *delegated* to role $A.r$. Any member E of role $B.s$ can further delegate role $A.r$ to a role $C.t$ (administered by C). The corresponding extension credential is denoted by $E \xrightarrow{A.r} C.t$.

We introduce attribute constraints to role and delegation credentials. Attribute constraints are assigned in role credentials by role administrators, or in delegation credentials by delegators. For example, a resource owner specifies that the number of times a delegation privilege can be further extended is at most two. We express attribute constraints on the arrow of a delegation expression, e.g. $A \xrightarrow{A.r\ [attr=v]*} B.s$, where * is a wildcard representing that there is zero or more attribute constraints following $A.r$. In our combined RBCD (cRBCD) model, we define boolean attribute *isPropagatable* for delegation transactions. It specifies whether or not a credential can be used to construct a valid delegation chain by credential discovery algorithm. If yes, then *isPropagatable* is set to true, else *isPropagatable* is set to false. In cRBCD model, only when *isPropagatable* is set to true by an original delegator, can a dynamically discovered credential be used to construct a valid delegation chain. Note that a credential with *isPropagatable* being false can still be extended. We discuss this security implication in details in Section 7.

We introduce predicates for our language model to express requirements for attributes of a delegatee or a delegation chain. Each predicate restricts the scope of delegation recipients. For example, the ranking of a delegatee's role should be greater than or equal to *senior engineer*. We express a predicate as $P(attr, v)$, where *attr* is the name of an attribute of delegation recipients, v is a value, and P is an evaluation function such as greater-than and less-than. A predicate is evaluated to either true or false. It may specify a constraint based on a literal value, as in $>(attr, 3)$, or on the value of another attribute, as in $>(attr_1, attr_2)$. All predicates in a credential must be satisfied in order for the credential to be valid. Predicates are written on the right of delegation expression as in $A \xrightarrow{A.r} B.s\ [P(attr, v)*]$, where * is a wildcard and represents that there is zero or more predicates following $B.s$.

Delegation predicates and constraints are used throughout this paper. Concrete examples are given in the next Section. Formal descriptions of their usage in our new RBCD protocols are presented in Section 4.1 and 5.2.

3 Example scenarios

In this section, we use examples to illustrate the use of merging multiple delegation chains and the use of predicates and constraints. Let us suppose that there is a collaboration project between a health center C and a clinic L. To facilitate the collaboration, the health center initiates a delegation chain and delegates its role

$C.guest$ to the role *L.manager* at the clinic. However, C imposes the following constraints on this delegation: the delegation chain can be further extended for at most four times, and the rank of delegation recipient needs to be at least second-level manager $M2$. We now express these restrictions in credential (1). Predicate $[\geq(\text{rank}, M2)]$ is true if rank $\geq M2$, and false otherwise.

$$C \xrightarrow{C.guest\ [\text{step}\leq 4]} L.manager\ [\geq(\text{rank}, M2)] \tag{1}$$

Credential (1) means that a member of *L.manager* with rank $\geq M2$ is also a member of $C.guest$. By the definition of RBCD, (1) can be further extended by members of *L.manager* to other entities. Bob is a second-level manager at L. He delegates role $C.guest$ to members of $A.doctor$ at hospital A. To limit the scope of delegation recipients, he specifies the following: constraint [steps ≥ 3] means that the delegated privilege can only be further extended for at most three times; predicate $[\geq(\text{rank}, D2)]$ requires a delegatee's rank to be at least $D2$. Alice wants to access some data controlled by center C. She is a doctor in hospital A with rank $D2$.

To prove that she has the health center C's delegated *guest* role, Alice obtains the delegation credential for role $A.doctor$ from a credential server, and groups it with her role credential. This gives credential (2).

$$\begin{aligned}&(C \xrightarrow{C.guest\ [\text{step}\leq 4]} L.manager\ [\geq(\text{rank}, M2)]), \quad (L \xrightarrow{L.manager\ [\text{rank}=M2]} \text{Bob}),\\ &(\text{Bob} \xrightarrow{C.guest\ [\text{step}\leq 3]} A.doctor\ [\geq(\text{rank}, D2)]), \quad (A \xrightarrow{A.doctor\ [\text{rank}=D2]} \text{Alice})\end{aligned} \tag{2}$$

By RBCD definition, credential (2) includes credential (1), Bob's affiliated role credential, the extension credential generated by Bob, and Alice's role credential. (2) and a signature signed with Alice's private key yield a complete delegation credential for Alice. Resource owner C verifies signatures and evaluates delegation predicates. In our example, predicates and constraints in (2) are all satisfied.

Now, let us use the above scenario to further illustrate how multiple delegations are merged into one delegation. Suppose that role $A.doctor$ in hospital A is not only delegated role $C.guest$, but also delegated role $M.member$ by a medical service M and role *P.consultant* by a company P as shown in credential (3) and (4), respectively. Assume that delegation (3) and (4) do not have attribute constraints.

$$M \xrightarrow{M.member} A.doctor \quad (3) \qquad\qquad P \xrightarrow{P.consultant} A.doctor \quad (4)$$

Suppose that an emergent medical operation requires the collaboration with poison experts at hospital H_1 and H_2. These experts, represented by role H_1*.expert* and H_2*.expert*, have to be granted role $C.guest$, and the privileges associated with credential (3), and (4). One approach is for Alice to extend each of credential (2), (3), and (4) to H_1*.expert* and H_2*.expert*, respectively. In comparison, a more efficient way is shown next. Alice creates a new local role with an arbitrary name $Alice.local$, and extends delegation of $C.guest$ to role $Alice.local$ in credential (5).

$$\begin{aligned}&(C \xrightarrow{C.guest\ [\text{step}\leq 4]} L.manager\ [\geq(\text{rank}, M2)]), \quad (L \xrightarrow{L.manager\ [\text{rank}=M2]} \text{Bob}),\\ &(\text{Bob} \xrightarrow{C.guest\ [\text{step}\leq 3]} A.doctor\ [\geq(\text{rank}, D2)]), \quad (A \xrightarrow{A.doctor\ [\text{rank}=D2]} \text{Alice}),\\ &\qquad (Alice \xrightarrow{C.guest} Alice.local)\end{aligned} \tag{5}$$

As shown in Figure 1, Alice also extends delegation (3) and (4) to role $Alice.local$ in credential (6) and (7), respectively. Then, she delegates $Alice.local$ to role H_1*.expert* at hospital H_1 in extension credential (8). Similarly, Alice delegates $Alice.local$ to role H_2*.expert*. We call this delegation process MERGE EXTEND operation. MERGE EXTEND can significantly reduce the number of credentials needed to be generated by Alice. The schematic drawing of credentials received by role H_1*.expert* is shown in Figure 1. To access a resource controlled by health center C, a member of role H_1*.expert* submits a proof consisting of credential (5) and (8). The former authorizes role $C.guest$ to $Alice.local$, and the latter proves that $Alice.local$ is delegated to role H_1*.expert*. Therefore, a member of H_1*.expert* is granted access as role $C.guest$.

Due to space limit, we do not give examples of using credential chain discovery in RBCD. Next, we present our mRBCD protocol that supports MERGE EXTEND operation and the use of delegation constraints and predicates.

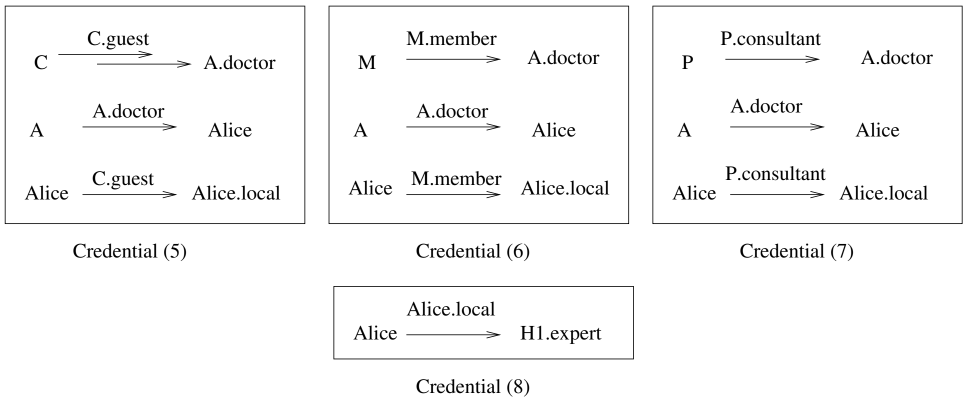

Figure 1. Components of the delegation credential for role *expert* at hospital H_1 issued by Alice. Double-arrow in credential (5) represents that role $C.guest$ is delegated to $A.doctor$ through intermediate delegator (Bob in our example). Constraints and predicates are omitted.

4 Merging delegation chains

In this section, we describe a RBCD protocol that supports operation MERGE EXTEND. This operation allows privileges associated with multiple delegation credentials $\{C\}$ to be extended into a new *local* role, which can then be delegated in just one extension credential. MERGE EXTEND reduces the number of delegation credentials to be issued, when there are multiple roles to be delegated to. A delegator D runs MERGE EXTEND algorithm as follows. D first creates a new local role, for example $D.local$. Then, D extends his delegation credentials to role $D.local$. The resulting delegation credentials are denoted as the credential set $\{C\}$. Finally, D delegates role $D.local$ to delegatees, by initiating a new delegation chain which is called an *intermediate delegation chain*. D also gives credentials $\{C\}$ to delegatees for verification purpose, so that delegations defined in $\{C\}$ can be linked to role $D.local$ and verified. We refer this role-based delegation protocol as *merge* RBCD (mRBCD) protocol. Details of mRBCD protocol with MERGE EXTEND operation are defined next.

4.1 mRBCD protocol

The mRBCD protocol has five operations: INITIATE, EXTEND, MERGE EXTEND, PROVE, and VERIFY. Operation MERGE EXTEND is the new operation that handles *merging* of delegation chains. All operations support the use of delegation constraints and predicates.

- INITIATE(P_{D_0}, s_{D_0}, $D_0.priv$, $A_1.r_1$, P_{A_1}, $[Cons]*$, $[Pred]*$):
  This operation is run by the resource owner D_0 of a privilege $D_0.priv$ to delegate $D_0.priv$ to an affiliated role $A_1.r_1$. It initiates a delegation chain for privilege $D_0.priv$, and is similar to INITIATE operation of the original RBCD protocol in [21]. The difference is that here a delegator is allowed to specify predicates and constraints to refine the scope of delegation recipients. Inputs include public key P_{D_0} of entity D_0, corresponding private key s_{D_0}, delegated privilege $D_0.priv$, role name $A_1.r_1$, public key P_{A_1} of role administrator A_1, and zero or more attribute constraints $[Cons]*$ and predicates $[Pred]*$ for the delegation. Output is a partial delegation credential C_1 for role $A_1.r_1$ signed with private key s_{D_0}. Credential C_1 is represented as $D_0 \xrightarrow{D_0.priv\ [Cons]*} A_1.r_1[Pred]*$. INITIATE can be modified and used for generating role credentials with restrictions by a role administrator. This is done by certifying the public key of a role member, and signing with the role administrator's private key.
- EXTEND(s_{D_n}, $D_0.priv$, C_n, R_{D_n}, $A_{n+1}.r_{n+1}$, $P_{A_{n+1}}$, $[Cons]*$, $[Pred]*$):
  This operation is similar to EXTEND operation of the original RBCD protocol. It is run by an intermediate delegator D_n to extend delegated privilege $D_0.priv$ to role $A_{n+1}.r_{n+1}$. Inputs are private key s_{D_n} of delegator D_n, delegated privilege $D_0.priv$, partial delegation credential C_n that gives privilege $D_0.priv$ to role $A_n.r_n$, role credential R_{D_n} of D_n, role name $A_{n+1}.r_{n+1}$, public key $P_{A_{n+1}}$ of role administrator A_{n+1}, attribute assignments $[Cons]*$, and predicates $[Pred]*$ specified by D_n. An extension credential denoted by

$(D_n \xrightarrow{D_0.priv\ [Cons_n]*} A_{n+1}.r_{n+1}\ [Pred_n]*)$ is generated and signed with private key s_{D_n}. The final output is a partial delegation credential C_{n+1}, which is a function of the credential C_n, the role credential R_{D_n} denoted by $(A_n \xrightarrow{A_n.r_n\ [Cons'_n]*} D_n\ [Pred_n]*)$, and the extension credential. Credential C_{n+1} may simply be the delegation credential C_n together with two individual credentials.

- MERGE EXTEND(s_D, R_D, $\{C\}$, $A.r$, P_A, $[Cons]*$, $[Pred]*$):
  This algorithm is run by an entity D to delegate privileges associated with a set of credentials $\{C\}$ to role $A.r$. The inputs include private key s_D of entity D, a role credential R_D of D, a set of credentials $\{C\}$ that are issued to the role of D, a delegation recipients' role $A.r$, public key P_A of role administrator A of role $A.r$, and constraints $[Cons]*$ and predicates $[Pred]*$ of attributes.
  Delegator D first locally creates a new role *D.local*. For each credential $C_j \in \{C\}$, D extends the delegated privilege $priv_j$ in C_j to the new role *D.local* by running EXTEND(s_D, $priv_j$, C_j, R_D, $A.r$, P_A, null, null) method above. (Two null arguments represent that no constraints and predicates are defined.) EXTEND returns a credential C'_j that delegates privilege $priv_j$ to role *D.local*. The set of credentials returned by EXTEND operations is denoted as $\{C'\}$. Next, delegator D initiates an intermediate delegation chain that delegates role *D.local* to role $A.r$, by running INITIATE(P_D, s_D, *D.local*, $A.r$, P_A, $[Cons]*$, $[Pred]*$) algorithm to generate a delegation credential C_r. C_r corresponds to a delegation chain for role *D.local*, which is called an *intermediate delegation chain*. The outputs of this operation are credential set $\{C\}$ and credential C_r, which together are delegation credentials for members of role $A.r$.
- PROVE($s_E, V.priv, R_E, \{C'\}$):
  This algorithm is run by a requester E with role credential R_E, who wants to access the privilege *V.priv* controlled by a resource owner V. Its main difference from PROVE of the original RBCD protocol [21] is that a proof produced by this algorithm may be constructed from two or more delegation chains, as a result of MERGE EXTEND operations.
  Inputs to this operation are private key s_E of entity E, role credential R_E, and a set of delegation credentials $\{C'\}$ that collectively form a delegation path between the resource owner and the requester E. $\{C'\}$ contains one delegation chain initiated by resource owner, and zero or more intermediate delegation chains initiated by intermediate delegators [1]. The operation produces a proof F that proves: (1) the role of E is delegated privilege *V.priv*, (2) E is a valid role member, and (3) E can authenticate his public key.
- VERIFY(F):
  This operation is performed by a resource owner V on proof F, which is submitted by a requester. If proof F does not involve any local roles, i.e. none of delegation certificates in F is generated by MERGE EXTEND, then VERIFY algorithm of the original RBCD protocol [21] is called. Otherwise, V authenticates proof F as follows.
  V first verifies delegation credentials $C_1, \ldots, C_n$ in F, each of them representing a delegation chain. Role and extension credentials that constitute each delegation chain are verified, and the satisfaction of delegation predicates is checked. Then, correct *linkages* between delegation chains are verified. That is, F should contain the proof of the following: (1) credential C_1 is issued by resource owner V, delegated privilege in C_1 is the requested privilege, and credential C_n is issued to requester E's role; (2) for $i \in [2,n]$, the delegated privilege in C_i is the last role (a local role) that receives delegation in credential C_{i-1}; (3) for C_i where $i \in [2,n]$, the original delegator is a valid member of the second to last role that receives delegation in credential C_{i-1} (for example in Figure 1, Alice is a member of *A.doctor*). Finally, proof F also allows the resource owner to verify the role membership and the public key of the requester.

The newly added MERGE EXTEND operation allows an intermediate delegator to initiate an intermediate delegation chain. If the number of delegation credentials to be extended is n and the number of roles receiving the

[1]The number of intermediate delegation chains equals the number of times operation MERGE EXTEND is run.

delegations is m, then MERGE EXTEND reduces the number of credentials to be generated from $n \times m$ to $n+m$. In Section 7, we discuss efficiency and implementation issues of mRBCD protocol. Next, we describe an approach that improves the flexibility of role-based cascaded delegation in distributed environment.

5 Credential chain discovery in RBCD

Credential chain discovery by Li, Mitchell, and Winsborough [16] facilitates dynamic delegation chain construction in decentralized environment. Delegation credentials do not need to be accumulated as in the RBCD model [21], and a delegation can be used to construct multiple delegation chains. However, the discovery process incurs communication costs, as it retrieves credentials from distributed credential servers. The costs are avoided in RBCD model. Nevertheless, the ability to construct dynamic delegation chain is important for a decentralized delegation model. In order to improve the delegation flexibility of RBCD models, we propose an integrated approach that combines credential chain discovery with RBCD protocol. At verification, credential chain discovery is used to collect RBCD credentials to construct a longer delegation path. Through the longer delegation path, trust between the requester and resource owner can be established, while this may not be possible through individual RBCD credentials alone. An example is shown in Figure 2. Compared to constructing delegation chains from single credentials as in the existing credential discovery framework [16], using RBCD credentials as building blocks can greatly reduce the costs associated with the discovery process. We call this integrated model as *combined* RBCD (cRBCD).

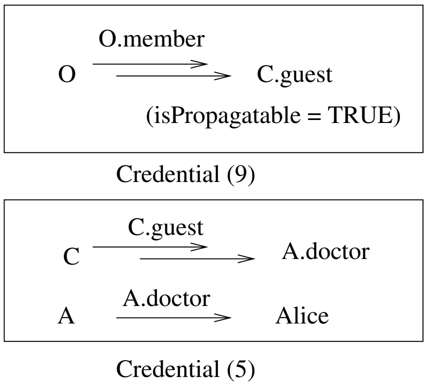

Figure 2. A delegation chain consists of Alice's credential (5), and credential (9) which is retrieved by credential chain discovery. Because *isPropagatable* is true, Alice is authorized role *O.member*. For simplicity, we use double-arrows to omit any intermediate delegators.

5.1 Discovery algorithm

We have modified the credential chain discovery algorithm by Li, Mitchell, and Winsborough [16] to improve the performance of discovery process and the security of shared resources. First, we do not require credential servers of intermediate delegators to relay credential query results from one server to another. They only need to response to credential queries submitted by a requester or a resource owner. Second, by setting boolean attribute *isPropagatable* to false in a credential, a resource owner chooses not to accept a delegation chain that is constructed by discovery algorithms. This effectively reduces the number of users who can access the resource. We further explore the security implications of this constraint in Section 7.

Next, we give a breath-first delegation chain discovery algorithm that is run by a requester E. It takes an initial set of E's credentials $\{C\}$ and a target privilege $V.priv$, and outputs a set of delegation credentials $\{C'\}$ that collectively authorizes $V.priv$ to E by resource owner V. The algorithm assumes that credentials are stored by delegation receivers (or their credential servers). Let X be a global set X for storing discovered credentials.

Discover($\{C\}, V.priv$):

1. For each delegation credential C_i in $\{C\}$, let the *original delegator* be O_i and the delegated privilege be role $O_i.r_i$. Requester E queries the credential server of delegator O_i to retrieve any credential at O_i that are issued to role $O_i.r_i$.

2. If nothing is retrieved by any of O_i's servers, then the algorithm returns an empty set since there are no valid delegation chains connecting E's roles.
3. Otherwise, E adds delegation credentials $\{C\}$ to set X. For each delegation credential returned by O_i's servers, E checks if the delegated privilege is $V.priv$. If yes, then a delegation chain is successfully discovered. Denote the credential that delegates privilege $V.priv$ as C_0. Construct a delegation chain between requester E and resource owner V from C_0 and credentials stored in X. This can be done by iteratively finding delegation credential $C_i \in X$ whose delegated privilege is the same as the last role that receives delegation in credential C_{i-1}. The iteration terminates when the last role that receives delegation in credential C_i is one of the requester E's. Credentials in the delegation chain are returned. If no credential returned by O_is delegates $V.priv$, requester E resets $\{C\}$ to be the delegation credentials retrieved from O_i's servers. Then repeats the discovery process by running $\mathsf{Discover}(\{C\}, V.priv)$.

The algorithm can also be run by resource owner V. The algorithm run by a requester is likely to perform better, because it processes a smaller number of credentials. Next, we present the cRBCD protocol that makes use of the discovery algorithm.

5.2 cRBCD protocol

The cRBCD protocol has four operations: INITIATE, EXTEND, PROVE, and VERIFY. MERGE EXTEND operation of mRBCD in Section 4 can also be added to cRBCD to support merging credentials at delegation. Since it is the same as in mRBCD protocol, we do not repeat it here. In cRBCD protocol, an original delegator specifies a boolean attribute *isPropagatable* in operation INITIATE, and credential chain discovery algorithm is used to dynamically construct delegation chains. The cRBCD protocol is defined as follows.

- INITIATE(P_{D_0}, s_{D_0}, $D_0.priv$, $A_1.r_1$, P_{A_1}, $[Cons]*$, $[Pred]*$, *bool*):
  This operation is similar as in operation INITIATE of mRBCD protocol in Section 4.1, and is run by a resource owner D_0 to initiate a delegation chain. The difference is that here the resource owner specifies whether or not this delegation of privilege $D_0.priv$ is allowed to be propagated to other roles at credential chain discovery, by setting attribute *isPropagatable* to the input boolean variable *bool*.
- EXTEND(s_{D_n}, $D_0.priv$, C_n, R_{D_n}, $A_{n+1}.r_{n+1}$, $P_{A_{n+1}}$, $[Cons]*$, $[Pred]*$):
  This operation is the same as EXTEND operation in RBCD protocol of Section 4.1.
- PROVE($s_E, V.priv, R_E, \{C\}$):
  This algorithm is run by requester E for proving that he is delegated privilege $V.priv$, where V is a resource owner. The inputs are private key s_E of entity E, privilege $V.priv$, a role credential R_E of E, and a set of delegation credentials $\{C\}$ issued to E's role. For each credential in $\{C\}$, check if any of them delegates privilege $V.priv$. If yes, construct and output a proof F by running PROVE($s_E, V.priv, R_E, \{C\}$) of mRBCD protocol in Section 4.1. Otherwise, run algorithm $\mathsf{Discover}(\{C\}, V.priv)$ in Section 5.1. If Discover returns an empty set, then no valid delegation chain can be found and an empty set is returned. If Discover returns non-empty credential set $\{C'\}$, then output a proof F that contains $\{C'\}$, role credential R_E, and a signature signed with s_E on a nonce chosen by the verifier.
- VERIFY(F):
  Similar to VERIFY algorithm in mRBCD protocol, resource owner examines proof F to authenticate a delegation chain. However, not every delegation credential can be used to construct a delegation chain in cRBCD, due to the constraints on delegation attribute *isPropagatable* on each credential. Let $C_1, \ldots, C_n$ be delegation credentials in proof F, each of them representing a delegation chain. Delegation credential C_1 is issued by resource owner V, delegated privilege in C_1 is the requested privilege, and C_n is issued to E's role. For $C_1, \ldots, C_{n-1}$, only delegation credentials with *isPropagatable* being true can be used to construct a delegation chain.

V first authenticates delegation credential C_1 through C_n and role credential of E, by verifying signatures and evaluating predicates. Then, the correct *linkages* between delegation chains are checked: delegated privilege in C_i (for $2 \leq i \leq n$) is the same as the last role receiving delegation in credential C_{i-1}. For example in Figure 2, $C.guest$ is the last role on the delegation chain in credential (9). It is also the delegated privilege in credential (5). Finally, proof F also allows the resource owner to verify the role membership and the signature of the requester.

Credential chain discovery retrieves independently issued credentials and dynamically constructs delegation chains. Besides that, it helps reduce the number of credentials carried by delegatees. An intermediate delegator D_n can choose *not* to run EXTEND to extend a delegation. Instead, D_n initiates a new delegation chain C' (by running INITIATE) that delegates his role $A_n.r_n$ to role $A_{n+1}.r_{n+1}$ with proper delegation constraints. Credential chain discovery retrieves credential C_n from credential server of intermediate delegator D_n. C_n along with C' give authorization to members of role $A_{n+1}.r_{n+1}$. In Section 7, we compare delegation chains contructed this way with ones contructed by running EXTEND operations.

6 Prototype Implementation

We have built a prototype of RBCD protocol in Java and C/C++. In this section, we describe the components of our implementation, and analyze our experimental results. Our implementation can be divided into several layers. The bottom layer is the implementation of a signature scheme. We deploy a C++ implementation of the Boneh-Lynn-Shacham (BLS) short signature scheme [10] written by Barreto [3, 4]. The second to bottom layer provides an Java interface for the functionalities in the C++ signature implementation. The next layer above is the Java implementation of a certificate system which uses BLS short signatures. Our certificate, public key, private key, and signature classes extend the ones in the java.security library, which facilitates their integration into other Java applications. The top layer of our implementation provides Java classes for the RBCD credential and operations.

6.1 Signature scheme implementation

The RBCD protocol can be implemented using any signature scheme. As argued in [21], signatures and public keys with short length help reduce the size of RBCD credentials, and compact credentials improve the efficiency of credential accumulations in RBCD. Towards this goal, we use the BLS short signature scheme [10] as our signing algorithm for credentials in our prototype. A 170-bit BLS short signature gives security equivalent to a 1024-bit RSA signature. In addition, public keys in BLS are the same length as signatures, which are significantly smaller than comparable RSA public keys. We use the existing C++ implementation of BLS short signature by Paul Barreto [3, 4], which implements fast pairing computation and allows fast signature verification. The signature implementation works on a specific supersingular elliptic curve ($y^2 = x^3 + x + 1$) over a specific finite field ($\mathbb{F}_{3^{97}}$). It uses Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL) [18] for big number operations, including elliptic curve arithmetics, cryptographic hashing, etc.

We build a bridging layer above the C/C++ signature code, in order to use java.security library in our implementation. Code in this layer is written in C++ and Java, and serves as a Java wrapper of signature operations including key generation, signing, and verification. This layer provides the transparency of C++ signature implementation for users running Java applications. To achieve this, we execute C++ signature executables as command line operations inside Java programs, which pass arguments for signature operations and interprets outputs of command line executions.

6.2 RBCD credential and operations

Our Certificate class uses BLS short signatures, and extends the Certificate abstract class in java.security.cert library. There are two types of certificates in our prototype. One is delegation certificate, and the other is role certificate. They have different contents, as explained in the INITIATE operation of Section 4.1. Both types of certificates contain objects of class Role, whose fields include a string of role name and the public key of the role administrator. Signing and verifying a certificate are easy in our implementation, which follows the standard Java API specifications. We implement PublicKey and PrivateKey interfaces in java.security for our BLS public and private keys. Signatures in this prototype are instances of our class BLSSignature, which provides sign and verify methods. Our implementation provides serialization methods for certificate objects, key objects, and signature objects.

At the top layer of our implementation are classes for RBCD credential and operations. RBCDCredential class extends the class Certificate in java.security.cert. A RBCDCredential object contains a linked list of certificate objects as its field. The certificates include both delegation certificates and role certificates, which are used to authenticate delegation chains. Methods in RBCDProtocol class follow the input/output specifications in the original RBCD protocol [21]. Method initiate in RBCDProtocol instantiates a RBCDCredential object by generating a delegation certificate signed by an original delegator. Method extend in RBCDProtocol appends new certificates from an intermediate delegator to an existing RBCD delegation credential, and outputs the new RBCDCredential object for delegatees. Method prove is called by a requester to produce an RBCDProof object, which contains a signature signed by the requester on a nonce and a RBCDCredential object. In method verify, a caller inputs a RBCD credential, which is verified and the consistency of public keys and privilege specifications among the certificate statements are checked.

Our prototype provides an infrastructure for RBCD implementation. It contains basic components required to realized an interoperatable cascaded delegation protocol. There are many immediate improvements to make, including supports for extensions proposed in this paper and for the use of aggregate signatures [9]. Short signature with aggregation ability can further reduce the size of our credentials, as described in [21]. Towards a full integration of RBCD implementation with java.security specifications, a Provider subclass needs to be defined to standardize our cryptographic operations.

7 Experimental results and discussion

In this section, we present and analyze our experimental results on our RBCD implementation. We also discuss the efficiency and security implications brought by our mRBCD and cRBCD extensions.

7.1 Experimental results

Our experiments are performed on a machine that has Linux operating system, 2Ghz AMD Athlon 64 Processor, and 512 MB main memory. Experimental results are shown in Figure 3. Our prototype is a standalone application, hence experimental results do not involve network transmission time. Running time data does not include generation time for key pairs and role certificates. For each RBCD credential of length one to thirty, we give the running time of generating and verifying the delegation credential in Figure 3. The y-values in Figure 3 represent the base-2 logarithms of milliseconds. Logarithmic scale is used for the easy visualization of data. For comparison, we also compute the total amount of time spent on generating and verifying BLS signatures in a credential. From Figure 3, we compute the average time of performing RBCD operations in Table 1. The average time for RBCD VERIFY operation and *Verifying signatures* in Table 1 represent the time required for verifying a RBCD credential of length one, and for signature verification for that credential, respectively.

All four sets of running time data in Figure 3 are linear in the length of a delegation chain. Generating a RBCD credential of length n requires one INITIATE operation and $n-1$ EXTEND operations, and is quite fast. The average time for a single INITIATE or EXTEND operation is 8.8 ms, as shown in Table 1. PROVE operation has similar average running time as INITIATE and EXTEND operations, because they all involve computing a BLS signature and regrouping certificates. For a RBCD credential, BLS signature generation takes a small portion of

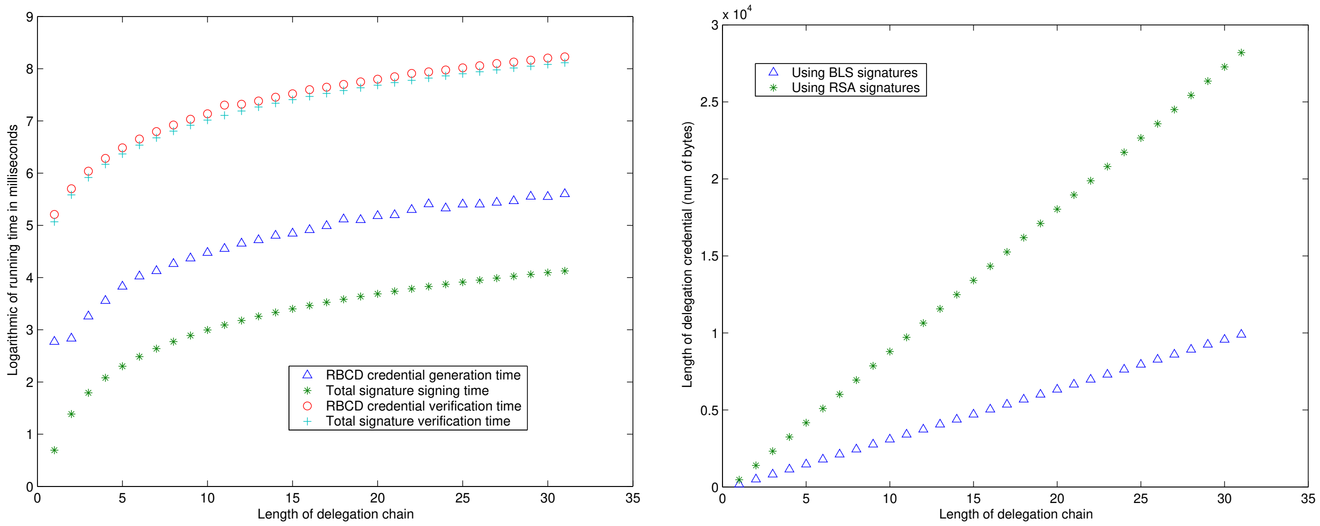

Figure 3. Experimental results in our implementation. In both figures, x-axis is the number of delegation extensions in a RBCD credential, i.e. number of delegators. The left chart gives the base-2 logarithm of the total running time in milliseconds of generating a RBCD credential of a specific length ('△'), of verifying a RBCD credential ('o'). It also gives the collective running time of computing BLS signatures in a RBCD credential ('*'), and of verifying BLS signatures in a RBCD credential ('+'). In the right chart, sizes of RBCD credentials signed with BLS signatures are shown in '△' symbols, and '*' symbols show the sizes of RBCD credentials signed with equivalent RSA signatures. Credential size is represented as the number of its bytes.

the total time. Table 1 shows signing a signature only takes 2 ms. The time difference between 8.8 ms and 2 ms is accounted for the work for initializing and updating objects and fields, I/O operations, overhead for running Java.

Operation	INITIATE/EXTEND	PROVE	*Signing a signature*	VERIFY	*Verifying signatures*
Average running time	0.0088 s	0.0080 s	0.0020 s	0.1234 s	0.1095 s

Table 1. Comparison of average running time in seconds.

Verifying a RBCD credential of length n is time-consuming. It requires authenticating n extension credentials, n role credentials, verifying the requester's signature on the nonce, and checking for consistency among credential statements. The average running time for verifying a delegation chain of length one is 0.1234 seconds. Most of the time is spent on signature verification, which takes on average 0.1095 seconds to complete as shown in Table 1. The average time of VERIFY and *verifying signatures* is computed as the average of the total running time divided by delegation length from length one to thirty. Therefore, 0.1095 seconds is the time of verifying about two BLS signatures [2]. Besides signature verification, consistency of delegation privilege and public keys among certificates is also checked, which increases the running time of VERIFY operation.

BLS short signatures are effective in reducing size of RBCD credentials. Signatures and public keys in our implementation are 160 bit long. The number of bytes of RBCD credentials of length one to thirty are counted and shown in Figure 3. We also compute and show the sizes of the same credentials using RSA signatures with equivalent security. Such a RSA signature is 960 bit long [3] and corresponding public key is more than 960 bits. It is easy to see the BLS signatures significantly reduces the size of RBCD credentials, compared to RSA signatures.

7.2 Security

The security of our RBCD models are based on the unforgeability of BLS short signature scheme [10]. We allow adversaries to observe the traffic on the network. They can also participate delegation processes, that is, an

[2] 2.0625 to be exact in our experiment.

[3] This is because the security multiplier is six in our BLS setup [4].

adversary may be a valid member of a role who can issue delegations and submit access requests. In our RBCD protocols, an adversary cannot successfully submit access requests on behalf of others, in particular, in the name of other role members. Similarly, an adversary cannot successfully issue delegations on behalf of others, even if she obtains another entity E's role credential and delegation credentials issued to E. This is because an adversary cannot forge a valid signature of E. Our protocols are secure against replay attacks, because it is infeasible for an adversary to forge a valid BLS short signature on a verifier-chosen nonce signed with other member's private key. These imply that an adversary cannot obtain or delegate privileges that are not authorized to them.

In decentralized delegation systems, protection of resource security relies crucially on the trustworthiness of delegators. In the cRBCD model, this is particularly important because delegation chains are constructed *ad hoc*. Credential chain discovery gives more possibilities for an unknown requester to establish trust with a resource owner. In the mean time, our cRBCD model provides resource owners a way of restricting the scope of resource sharing by specifying an boolean attribute *isPropagatable*. A credential with *isPropagatable* set to false by an original delegator can still be extended by EXTEND operation. However, by definition it cannot be used by credential discovery to construct any delegation chain. There is a semantic difference between a delegation chain produced by EXTEND operations and a discovered delegation chain. The former delegates privileges only to members of an affiliated role, while the latter gives authorizations to members of both the affiliated and delegated roles. As defined in Section 2.1, RBCD delegation $A \xrightarrow{A.r} B.s$ is effective for members with affiliated role $B.s$, but does not automatically propagate the privilege $A.r$ to those who are delegated role $B.s$. This propagation is done by credential chain discovery. Hence, discovered delegation chain can give authorizations to more users than cascaded delegation. Therefore, *isPropagatable* attribute gives original delegators the ability of controlling the scope of resource sharing.

7.3 Efficiency

In this section, we analyze the efficiency and security properties of cRBCD and mRBCD protocols. Our generalized RBCD protocols provide flexibility of delegation and credential manipulation without introducing significant computation and communication overhead. In case where n delegation chains are to be extended to m roles, MERGE EXTEND operation in mRBCD improves delegation efficiency by reducing the number of credentials to be generated from $n \times m$ to $n + m$. As illustrated by our example in Section 3, the efficiency improvement can be important for forming dynamic collaboration in emergent medical situations. A simple data structure for implementing merged RBCD credentials is the linked list. Each RBCD credential is a linked list of individual certificates. MERGE EXTEND operation first links multiple linked lists that correspond to multiple RBCD credentials, and then extends the resulting linked list by appending new certificates from the intermediate delegator. Merged credential can also be easily reconstructed into individual delegation credentials, which is useful when the holder of the credential only wants to extend part of the delegated privileges to others.

The use of credential chain discovery in combination with cascaded delegation increases delegation flexibility in RBCD model, as delegations do not have to be issued in a top-down fashion from the original issuer. It allows independently issued credentials to be connected. cRBCD also reduces communication and transmission costs in credential chain discovery. It requires fewer queries to construct a delegation chain from RBCD credentials than from individual credentials, because some portion of a delegation chain is already captured in an RBCD credential, and does not need to be discovered from scratch.

8 Related work

In this section, we compare our work with related trust management frameworks and access control policy languages. X-GTRBAC Admin [6] is an administration model for policy administration within a large enterprise. It specifies the user-to-role and permission-to-role assignments in the XML-based Generalized Temporal Role

Based Access Control (X-GTRBAC) framework [14]. X-GTRBAC Admin supports decentralized administration by distributing assignment tasks to multiple domains within the enterprise while enforcing temporal constraints. In comparison, our RBCD models aim at the decentralized trust management among members of independent organizations. Therefore, X-GTRBAC Admin is complementary to RBCD models.

Permission-based delegation model (PBDM) built on RBAC supports user-to-user, role-to-role delegations [23]. A delegator creates one or more temporary delegation roles and assigns delegatees to particular roles. Delegations in PBDM requires changes of role hierarchies by the proper authority, for example, a project leader who has write access to the role assignment and access policies. PBDM does not address how decentralized delegations, where central authorities may not have to be involved.

The *RT* framework is a family of Role-based Trust management languages for representing policies and credentials in decentralized authorization [15]. Our cRBCD model in Section 5 combines the credential chain discovery algorithms of *RT* framework with the role-based cascaded delegation [21], which gives the original RBCD model more flexibility in constructing delegation chains. The PolicyMaker [8] and KeyNote [7] trust management systems authorize decentralized access by checking a proof of compliance. SPKI/SDSI (Simple Public Key Infrastructure/Simple Distributed Security Infrastructure) is a public-key infrastructure emphasizing decentralized name space and flexible authorization [11, 12]. KeyNote and SPKI/SDSI do not support role-based delegations, and assume credentials are centrally stored. Cascaded delegation [20, 22] schemes were proposed for proxy authentication and authorization. These models do not support delegations to roles, and the delegation credentials are not as compact as ours.

Predicates used to express constraints of attributes in our delegation model are similar to predicates in several access control policy languages [1, 5], and the Web Services Policy Language (WSPL) is one of them. WSPL is a policy language for specifying authorization, quality-of-service, privacy, etc [1]. The syntax is a subset of the eXtensible Access Control Markup Language (XACML) [17] Standard. A WSPL policy is a sequence of one or more rules, where each rule represents an acceptable choice for satisfying the policy and is a sequence of predicates. WSPL focuses on using predicates for comparisons of different policies, and we use predicates to restrict delegation scopes. Another access control policy language that extensively uses predicates is the language in Cassandra, which is a role-based trust management system [5]. Cassandra's policy specification language is based on Datalog with constraints, where properties asserted by credentials are predicates. Our language model aims at supporting the functionality of role-based delegation of RBCD, while predicates in Cassandra are also used for role hierarchy, separation of duties, etc.

9 Conclusions and future work

We have made several practical extensions to the original RBCD protocol [21] to improve its expressiveness, delegation flexibility, and efficiency. The mRBCD model supports the merging of multiple delegation chains and improves the delegation efficiency. The cRBCD model integrates credential chain discovery with the cascaded delegation. We have also presented our RBCD prototype written in Java and C++. For future work, we would like to give a formal semantic analysis of the differences between the *RT* framework [15] and the RBCD model. We plan to expand our current implementation to support extensions presented in this paper. Future work would also include generalization of the short signature implementation to aggregate short signatures, and using them to further shorten the size of RBCD credentials.

10 Acknowledgement

We are grateful to Paulo S.L.M. Barreto for sharing us his C++ implementation of BLS short signature scheme, and to William H. Winsborough for his helpful comments on the cRBCD model.

References

[1] A. H. Anderson. An introduction to the Web Services Policy Language (WSPL). In *POLICY '04*, pages 189–192, 2004.

[2] J. Bacon, K. Moody, and W. Yao. Access control and trust in the use of widely distributed services. In *Middleware 2001*, volume 2218 of *LNCS*, pages 295–310. Springer, 2001.

[3] P. S. Barreto. C++ implementation of BLS short signature scheme.

[4] P. S. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In *Advances in Cryptology — Crypto '02*, volume 2442 of *LNCS*, pages 354–368. Springer-Verlag, 2002.

[5] M. Y. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY 2004*, pages 159–168, 2004.

[6] R. Bhatti, J. Joshi, E. Bertino, and A. Ghafoor. X-GTRBAC admin: a decentralized administration model for enterprise wide access control. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT '04)*, pages 78–86, 2004.

[7] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Proceedings of Security Protocols International Workshop*, 1998.

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

[9] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in Cryptology — Eurocrypt '03*, pages 416–432, 2003.

[10] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology — Asiacrypt '01*, volume 2248 of *LNCS*, pages 514–532. Springer-Verlag, 2001.

[11] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[12] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Yloenen. Simple public key certificate. `http://www.ietf.org/rfc/rfc2693.txt`.

[13] C. Gentry. Certificate-based encryption and the certificate revocation problem. In *Advances in Cryptology — Eurocrypt '03*, volume 2656 of *LNCS*, pages 272–293, 2003.

[14] J. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Trans. Knowl. Data Eng.*, 17(1):4–23, 2005.

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

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

[17] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shahi. First experiences using XACML for access control in distributed systems. In *Proceedings of the ACM Workshop on XML Security 2003*, pages 25–37, October 2003.

[18] Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL). `http://indigo.ie/~mscott/`.

[19] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29, Number 2:38–47, 1996.

[20] K. R. Sollins. Cascaded authentication. In *Proceedings of 1988 IEEE Symposium on Security and Privacy*, pages 156–163, April 1988.

[21] R. Tamassia, D. Yao, and W. H. Winsborough. Role-based cascaded delegation. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT '04)*, pages 146 – 155. ACM Press, June 2004.

[22] V. Varadharajan, P. Allen, and S. Black. An analysis of the proxy problem in distributed systems. In *Proceedings of 1991 IEEE Symposium on Security and Privacy*, pages 255–275, 1991.

[23] X. Zhang, S. Oh, and R. Sandhu. PBDM: A flexible delegation model in RBAC. In *Proceedings of the ACM*

Symposium on Access Control Models and Technologies, pages 149 – 157. ACM Press, 2003.