Distributed Credential Chain Discovery in Trust Management *

Ninghui Li
Department of Computer Science
Stanford University
Gates 4B
Stanford, CA 94305-9045
ninghui.li@cs.stanford.edu

William H. Winsborough
Network Associates Laboratories
15204 Omega Drive
Suite 300
Rockville, MD 20850-4601
william_winsborough@nai.com

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

Abstract

We introduce a simple Role-based Trust-management language RT_0 and a set-theoretic semantics for it. We also introduce credential graphs as a searchable representation of credentials in RT_0 and prove that reachability in credential graphs is sound and complete with respect to the semantics of RT_0. Based on credential graphs, we give goal-directed algorithms to do credential chain discovery in RT_0, both when credential storage is centralized and when credential storage is distributed. A goal-directed algorithm begins with an access-control query and searches for credentials relevant to the query, while avoiding considering the potentially very large number of credentials that are unrelated to the access-control decision at hand. This approach provides better expected-case performance than bottom-up algorithms. We show how our algorithms can be applied to SDSI 2.0 (the "SDSI" part of SPKI/SDSI 2.0).

Our goal-directed, distributed chain discovery algorithm finds and retrieves credentials as needed. We prove that the algorithm is correct by proving that the algorithm is sound and complete with respect to the credential graph composed of the credentials it retrieves, and that the algorithm retrieves all credentials that constitute a traversable chain. We further introduce a storage type system for RT_0, which guarantees traversability of chains when credentials are well typed. This type system can also help improve search efficiency by guiding search in the right direction, making distributed chain discovery with large number of credentials feasible.

*An extended abstract of a preliminary version of this paper appeared in *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS-8)*, pages 156–165, ACM Press, November 2001.

1 Introduction

Several trust-management (TM) systems have been proposed in recent years to address authorization in decentralized environments, *e.g.*, SPKI/SDSI [6, 8], PolicyMaker [3, 4], KeyNote [2], Delegation Logic [14, 16]. These systems are based on the notion of delegation, whereby one entity gives some of its authority to other entities. The process of making access control decisions involves finding a chain of credentials that delegates the authority from the source to the requester. Thus, a central problem in trust management is to determine whether such a chain exists and, if so, to find it. We call this the *credential chain discovery problem*. This is different from the *certificate path discovery problem* for X.509 certificates [7], because credentials in TM systems generally have more complex meanings than simply binding names to public keys. A TM credential chain is often a graph, rather than a linear path. In this paper, we address the credential chain discovery problem (the *discovery problem* for short) in TM systems.

Almost all existing work addressing the discovery problem ([2, 4, 6, 14, 16]) assumes that one has already gathered all the potentially relevant credentials in one place and does not consider how to gather these credentials. The assumption that all credentials are stored in one place is at odds with the tenet of trust management. Since trust management is for decentralized control, systems that use trust management typically issue and often store credentials in a distributed manner. Distributed storage of credentials raises some nontrivial questions. When trying to construct a credential chain to answer an access-control query, where should one look for credentials? Often, one cannot look everywhere; in that case, when a chain cannot be found, how can one be sure that none exists? Distributed credential chain discovery requires a scheme to address these problems.

Distributed discovery also requires an evaluation procedure that is goal-directed, expending effort only on chains that involve the requester and the access mediator, or its trusted authorities, and considering credentials in a demand-driven manner. Goal-directed algorithms can be contrasted with bottom-up algorithms (*e.g.*, the algorithm in [6]), which require collecting all credentials before commencing evaluation. In the Internet, with distributed storage of millions of credentials, most of them unrelated to one another, goal-directed techniques will be crucial. Distributed credential chain discovery also requires a procedure that can begin evaluation with an incomplete set of credentials, then suspend evaluation, issue a request for credentials that could extend partial chains, then resume evaluation when additional credentials are obtained, and iterate these steps as needed.

In this paper, we address these problems associated with distributed chain discovery. As a concrete foundation, we introduce a trust-management language, RT_0, which is the first (and the simplest) language in the *RT* framework [17]. *RT* stands for Role-based Trust management; it is a family of languages for representing credentials and policies. We will discuss other elements of the *RT* framework briefly in Section 6. In the current paper, we provide goal-directed chain discovery algorithms based on a graphical representation of RT_0 credentials. This representation makes it straightforward to suspend and resume searches for multiple potential chains.

1.1 Motivating Examples

We now give more detailed motivations for studying distributed credential chain discovery through two examples. The first example involves a simple linear chain. The second illustrates what we call attribute-based delegation, and therefore involves a more complicated chain.

Example 1 A fictitious Web publishing service, EPub, offers a discount to preferred customers of its parent organization, EOrg. EOrg considers students of the university StateU to be preferred customers. StateU delegates the authority over identifying students to RegistrarB, the registrar of one of StateU's campuses. RegistrarB then issues a credential to Alice stating that Alice is a student. These are represented by four RT_0 credentials:

EPub.discount $\longleftarrow$ EOrg.preferred	(1)	EOrg.preferred $\longleftarrow$ StateU.student	(2)
StateU.student $\longleftarrow$ RegistrarB.student	(3)	RegistrarB.student $\longleftarrow$ Alice	(4)

The credential "EPub.discount $\longleftarrow$ EOrg.preferred" can be read as: if EOrg gives some entity the attribute "preferred", EPub gives the entity the attribute "discount". An alternative reading is that the set of entities that are entitled to EPub.discount includes the set of entities entitled to EOrg.preferred. In this credential, we call EPub the *issuer*, EOrg the *subject*, and "EOrg.preferred" the *body* of the credential. Similarly, for "RegistrarB.student $\longleftarrow$ Alice", we call RegistrarB the issuer and Alice the subject, as well as the body, of the credential. The four credentials above form a chain proving that Alice is entitled to a discount:

$$\text{EPub.discount} \overset{(1)}{\longleftarrow} \text{EOrg.preferred} \overset{(2)}{\longleftarrow} \text{StateU.student} \overset{(3)}{\longleftarrow} \text{RegistrarB.student} \overset{(4)}{\longleftarrow} \text{Alice}. \blacksquare$$

When one tries to prove that Alice is entitled to EPub's discount, each credential in the chain needs to be collected. The question we take up is where credentials should be stored to enable that collection. We say an entity A *stores* a credential if we can find the credential once we know A. Some other entity, such as a directory, may actually house the credential on A's behalf. Also, by storing a credential, we mean both storing and providing access to the credential. To be useful, a credential must be stored with its issuer or with its subject.

In Example 1, suppose that credential (1) is a local policy stored with EPub, and (4) is stored with Alice. One needs to be able to find (2) and (3). The most realistic arrangement is to have credential (2) stored with EOrg and (3) with RegistrarB.[1] The chain can then be discovered by searching for the credentials that compose the chain, starting from its two ends. Starting from Alice, one can find (4), thereby learning that one needs to go to RegistrarB to look for credentials involving RegistrarB.student. In this way, one can obtain (3). From the other direction, knowing (1), one can go to EOrg and find (2), thereby completing discovery of the chain shown above. No prior TM system supports discovery in this arrangement, probably because subject- and issuer-storage cannot be intermixed arbitrarily: if both (2) and (3) are stored exclusively with StateU, the chain cannot be found without trying to contact every entity in the system until, eventually contacting StateU, the credentials are obtained. In most decentralized systems, it is impossible or prohibitively expensive to enumerate all entities in the systems. For all practical purposes, if one cannot find a credential chain without contacting every entity, one cannot find it at all.

One approach to avoid storing both (2) and (3) only with StateU is to require that all credentials be stored with subjects. This enables one to find the credentials (4), (3), and (2) successively, thereby discovering the chain. A disadvantage of this arrangement is that it requires StateU to store all the credentials authorizing StateU's students for any resource. This makes StateU a bottleneck. Also, one can argue that (2) is really a local policy of EOrg; there should be no need for EOrg to give the credential to StateU. This point is further illustrated in Example 2 below.

Another approach is to require all credentials be stored with their issuers; then the chain can also be discovered. However, this approach is impractical for many applications. For instance, it would require credential (4) to be stored with RegistrarB. Requiring RegistrarB to be the sole

[1] EPub may obtain credential (2) from EOrg and cache it, so that it does not need to ask for the credential each time it tries to prove that someone is entitled to a discount. Similarly, Alice may cache credential (3). However, credentials (2) and (3) still need to be found in the first place.

provider of access to student registration credentials imposes a significant burden. We will discuss the problems with issuer storage of credentials further in Example 2 below.

A more reasonable scheme is to require each credential that has an entity as its body, *e.g.*, credential (4), to be stored with its subject, and all other credentials stored with issuers. This scheme would enable the chain in Example 1 to be found; however, this simple scheme does not work when credentials use more expressive features in trust-management systems. Some TM systems, such as SDSI and Delegation Logic, allow what we call *attribute-based delegation*, that is the delegation of attribute authority to entities having certain attributes. Attribute-based delegation is achieved in SDSI through linked names, and in Delegation Logic through dynamic threshold structures and through conditional delegations. Attribute-based delegation is illustrated in the following example.

Example 2 In Example 1, we have "EOrg.preferred $\longleftarrow$ StateU.student". This is a delegation from EOrg to StateU. Now say EOrg considers students of all universities to be preferred customers. Without attribute-based delegation, EPub has to know all the universities and delegate explicitly to each of them. Using attribute-based delegation, EOrg can delegate the authority over the identification of students to entities that EOrg believes are legitimate universities. Of course, EOrg can additionally delegate the authority over identifying universities to another entity, *e.g.*, a fictitious Accrediting Board for Universities, ABU. The following scenario revises the one in Example 1. Credentials (1), (3), and (4) are unchanged, while (2) is replaced by (5), (6), and (7).

EPub.discount $\longleftarrow$ EOrg.preferred	(1)		
StateU.student $\longleftarrow$ RegistrarB.student	(3)	RegistrarB.student $\longleftarrow$ Alice	(4)
EOrg.university $\longleftarrow$ ABU.accredited	(5)	ABU.accredited $\longleftarrow$ StateU	(6)
EOrg.preferred $\longleftarrow$ EOrg.university.student	(7)		

These credentials form a chain from Alice to EPub.discount, consisting of three parts.

$$\begin{aligned}
&\text{part(a): EPub.discount} \xleftarrow{(1)} \text{EOrg.preferred} \xleftarrow{(7)} \text{EOrg.university.student}\\
&\text{part(b): EOrg.university} \xleftarrow{(5)} \text{ABU.accredited} \xleftarrow{(6)} \text{StateU}\\
&\text{part(c): StateU.student} \xleftarrow{(3)} \text{RegistrarB.student} \xleftarrow{(4)} \text{Alice}
\end{aligned}$$

Here, part(b) shows that StateU has the attribute on the basis of which EOrg delegates to StateU the authority to identify students. In this way, it connects part(a) and part(c) into a chain.

Systems that support attribute-based delegation allow the ability to delegate to strangers whose trustworthiness is determined based on their own attributes. Attribute-based delegation promises flexibility and scalability; however, it significantly complicates the structure and discovery of credential chains. Example 2 illustrates several points. First, requiring all credentials to be stored with subjects is not reasonable. For instance, credential (5) is a local policy of EOrg, and of limited interested to ABU. In Example 1, one could argue that requiring StateU to store credential (2) is still plausible, since (2) authorizes students of StateU. Here, requiring ABU to store all credentials entrusting its judgement on universities is much less plausible. Second, the scheme in which all credentials are stored with their issuers, is impractical. Under such a scheme, in order to prove that Alice belongs to EOrg.university.student, one must obtain from ABU a complete list of universities, and then contact each university in turn to ask whether Alice is their student. Third, if all chains were linear, it might be reasonable simply to store with subjects exactly those credentials having entities as their bodies. However, Example 2 illustrates why this approach breaks down when we generalize to allow attribute-based delegation. Using this scheme, starting from EPub, one can find

(1), (7), and (5). Starting from Alice, one can find (4). However, since both (3) and (6) are stored only with StateU, we cannot find them. ∎

As we argued, distributed credential chain discovery requires goal-directed evaluation procedures. Even when all credentials are stored centrally, goal-directedness is still a big advantage, especially when the credential pool is very large. For example, in Example 2, suppose that we also have thousands of universities issuing credentials for millions of students. In order to find the chain from Alice to EPub.discount, a bottom-up algorithm, such as the one in [6], would have to consider each of the millions of credentials. By contrast, a goal-directed algorithm has the potential to avoid touching most of them.

1.2 Summary

The rest of this paper is organized as follows. In Section 2, we present the syntax and a set-theoretic semantics for RT_0, which supports attribute-based delegation and subsumes SDSI 2.0 (the "SDSI" part of SPKI/SDSI 2.0 [8]). In Section 3, we study chain discovery with centrally stored RT_0 credentials. We introduce the notion of credential graphs and present goal-directed, graph-based algorithms for centralized chain discovery in RT_0. We also show how to use our algorithms to perform goal-directed chain discovery for SDSI 2.0, for which existing algorithms are either not goal-directed, or do not support alternation between credential collection and evaluation steps. In Section 4, we extend the algorithms in Section 3 and present an algorithm to do chain discovery in RT_0 when credential storage is distributed. We also introduce notions of traversable chains and prove that traversable chains can be discovered by using the distributed search algorithm. In Section 5, we introduce a type system for credential storage and a notion of well-typed credentials; they constrain credential storage sufficiently to ensure chain traversability with well-typed credentials. This type system can also help improve search efficiency by guiding search in the right direction, making distributed chain discovery with large number of credentials feasible. In Section 6, we discuss future directions and related work. We conclude in Section 7. An appendix provides all proofs not contained in the paper's body.

2 RT_0: A Role-based Trust-management Language

The credentials presented in Examples 1 and 2 are RT_0 credentials. We now formally introduce RT_0. In Section 2.1, we present RT_0's syntax, discuss its intended meaning, and compare it to SDSI. In Section 2.2, we give a declarative semantics for RT_0.

2.1 The Syntax of RT_0

The basic constructs of RT_0 include entities and role names. *Entities* are also called principals in the literature. They can issue credentials and make requests. RT assumes that one can determine which entity issued a particular credential or request. Public/private key pairs clearly make this possible. In some environments, an entity could also be, say, a secret key or a user account. In this paper, we use A, B, and D, sometimes with subscripts, to denote entities. In RT_0, a *role name* is an identifier, say, a string. We use r, r_1, r_2, *etc.*, to denote role names.

RT_0 also has roles. A *role* takes the form of an entity followed by a role name, separated by a dot, *e.g.*, $A.r$ and $B.r_1$. The notion of roles is central in RT_0. A *role* in RT defines a set of entities who are members of this role. Each entity A has the authority to define who are the members of each role of the form $A.r$. A role can also be viewed as an attribute. An entity is a member of a role if and only if it has the attribute identified by the role. For example, we can view StateU.student both as a role (or a group of entities) and as an attribute. In RT_0, an access control permission is represented as a role as well. For example, the permission to shut down a computer can be represented by a role OS.shutdown.

There are four kinds of credentials in RT_0, each corresponding to a different way of defining role membership:

- *Type-1*: $A.r \longleftarrow B$

  A and B are (possibly the same) entities, and r is a role name.

  This means that A defines B to be a member of A's r role, *i.e.*, $\mathit{members}(A.r) \ni B$, where $\mathit{members}(A.r)$ represents the set of entities that are members of $A.r$. In the attribute-based view, this credential can be read as B has the attribute $A.r$, or equivalently, A says that B has the attribute r.

- *Type-2*: $A.r \longleftarrow B.r_1$

  A and B are (possibly the same) entities, and r and r_1 are (possibly the same) role names.

  This means that A defines its r role to include all members of B's r_1 role, *i.e.*, $\mathit{members}(A.r) \supseteq \mathit{members}(B.r_1)$. In other words, A defines the role $B.r_1$ to be more powerful than $A.r$, in the sense that a member of $B.r_1$ can do anything that the role $A.r$ is authorized to do. Such credentials can be used to define role hierarchies in Role-Based Access Control (RBAC) [19]. The attribute-based reading of this credential is: if B says that an entity has the attribute r_1, then A says that it has the attribute r. In particular, if r and r_1 are the same, this is a delegation from A to B of authority over r.

- *Type-3*: $A.r \longleftarrow A.r_1.r_2$

  A is an entity, and r, r_1, and r_2 are role names. We call $A.r_1.r_2$ a *linked role*.

  This means that $\mathit{members}(A.r) \supseteq \mathit{members}(A.r_1.r_2) = \bigcup_{B \in \mathit{members}(A.r_1)} \mathit{members}(B.r_2)$. The attribute-based reading of this credential is: if A says that an entity B has the attribute r_1, and B says that an entity D has the attribute r_2, then A says that D has the attribute r. If r and r_2 are the same, A is delegating its authority over r to anyone that A believes to have the attribute r_1. This is attribute-based delegation: A identifies B as an authority on r_2 not by using (or knowing) B's identity, but by another attribute B has (*viz.*, r_1).

- *Type-4*: $A.r \longleftarrow f_1 \cap f_2 \cap \cdots \cap f_k$

  A is an entity, k is an integer greater than 1, and for each $j \in [1..k]$, f_j is an entity, a role, or a linked role starting with A. We call $f_1 \cap f_2 \cap \cdots \cap f_k$ an *intersection*.[2]

  This means that $\mathit{members}(A.r) \supseteq (\mathit{members}(f_1) \cap \cdots \cap \mathit{members}(f_k))$.

A *role expression* is an entity, a role, a linked role, or an intersection. We use e, e_1, e_2, *etc*, to denote role expressions. By contrast, we use $f_1, \ldots, f_k$ to denote the intersection-free expressions occurring in intersections. All credentials in RT_0 take the form, $A.r \longleftarrow e$, where e is a

[2] In later design of the RT framework [17], an intersection can only contain roles. That restriction simplifies design and implementation, and does not change expressive power, since one can always add additional intermediate roles. Here, we adhere to the definition in [18].

role expression. Such a credential means that $\mathit{members}(A.r) \supseteq \mathit{members}(e)$, as we formalize in Section 2.2 below. We say that this credential *defines* the role $A.r$. (This choice of terminology is motivated by analogy to name definitions in SDSI, as well as to predicate definitions in logic programming.) We call A the *issuer*, e the *body*, and each entity in $\mathit{base}(e)$ a *subject* of this credential, where $\mathit{base}(e)$ is defined as follows: $\mathit{base}(A) = \{A\}$, $\mathit{base}(A.r) = \{A\}$, $\mathit{base}(A.r_1.r_2) = \{A\}$, $\mathit{base}(f_1 \cap \cdots \cap f_k) = \mathit{base}(f_1) \cup \cdots \cup \mathit{base}(f_k)$.

Readers familiar with Simple Distributed Security Infrastructure (SDSI) [6, 8] may notice the similarity between RT_0 and SDSI's name certificates. Indeed, our design is heavily influenced by existing trust-management systems, especially SDSI and Delegation Logic (DL) [16]. RT_0 can be viewed as an extension to SDSI 2.0 or a syntactically sugared version of a subset of DL. The arrows in RT_0 credentials are the reverse direction of those in SPKI/SDSI. We choose to use this direction to be consistent with an underlying logic programming reading of credentials and with directed edges in credential graphs, introduced below in Section 3. In addition, RT_0 differs from SDSI 2.0 in the following two aspects.

First, SDSI allows arbitrarily long linked names, while we allow only length-2 linked roles. There are a couple of reasons for this design. We are not losing any expressive power; one can always break up a long chain by introducing additional role names and credentials. Moreover, it often makes sense to break long chains up, as doing so creates more modular policies. If A wants to use $B.r_1.r_2.\cdots.r_k$ in its credential, then $B.r_1.r_2.\cdots.r_{k-1}$ must mean something to A; otherwise, why would A delegate authority to members of $B.r_1.r_2.\cdots.r_{k-1}$? Having to create a new role makes A think about what $B.r_1.r_2.\cdots.r_{k-1}$ means. Finally, restricting the length of linked roles simplifies the design of algorithms for chain discovery and the storage type scheme for guaranteeing distributed discovery. For the same reasons, we do not allow credentials of the form $A.r \longleftarrow B.r_1.r_2$ when $A \neq B$.

Second, SDSI does not have RT_0's type-4 credentials, and so RT_0 is more expressive than the current version of SDSI 2.0. Intersections and threshold structures (*e.g.*, those in [8]) can be used to implement one another. Threshold structures may appear in name certificates according to [8] and earlier versions of [9]. This is disallowed in [6] and the most up-to-date version of [9], because threshold structures are viewed as too complex [6]. Intersections provide similar functionality with simple and clear semantics.

Note that a type-2 credential $A.r \longleftarrow B.r_1$ can be represented using a type-1 credential and a type-3 credential: $A.r_0 \longleftarrow B, A.r \longleftarrow A.r_0.r_1$, in which r_0 is a new role name. We chose to include type-2 credentials nonetheless because of the following two practical considerations. First, conceptually, a type-2 credential represents a simple delegation, while a type-3 credential represents an attribute-based delegation, which is a more complicated notion. Second, we expect type-2 credentials are much more common than type-3 credentials and want to avoid having to use extra role names and credentials to represent them. Also note that although we restrict length of linked roles, we chose not to restrict the size of an intersection. We could require each intersection to be of the form $A.r \longleftarrow f_1 \cap f_2$; one can break up long intersections by introducing new role names and credentials. However, for RT_0, we find no semantic motivation or practical benefit for requiring this, unlike in the case of long linked names.

2.2 The Semantics of RT_0

This section presents a declarative semantics of RT_0. We begin by formalizing the syntactic categories of the language. RT_0 has a countable set of entities, denoted by *Entity*, and a countable set of role names, denoted by *RoleName*. The sets *Role*, *LinkedRole*, *Intersection*, and *RoleExpression* are respectively the sets of roles, linked roles, intersections, and role expressions over *Entity* and *RoleName*:

$$\begin{array}{rcl} \mathit{Role} & = & \{A.r \mid A \in \mathit{Entity},\ r \in \mathit{RoleName}\} \\ \mathit{LinkedRole} & = & \{A.r_1.r_2 \mid A \in \mathit{Entity},\ r_1, r_2 \in \mathit{RoleName}\} \\ \mathit{Intersection} & = & \{f_1 \cap \cdots \cap f_k \mid f_i \in \mathit{Entity} \cup \mathit{Role} \cup \mathit{LinkedRole}\} \\ \mathit{RoleExpression} & = & \mathit{Entity} \cup \mathit{Role} \cup \mathit{LinkedRole} \cup \mathit{Intersection} \end{array}$$

Definition 1 (Set-theoretic Semantics, $\mathcal{S}_\mathcal{C}$) For a given set[3] $\mathcal{C}$ of RT_0 credentials, the semantics of $\mathcal{C}$, $\mathcal{S}_\mathcal{C}$, is a function mapping roles to sets of entities: $\mathcal{S}_\mathcal{C} : \mathit{Role} \to \wp(\mathit{Entity})$, where $\wp(\mathit{Entity})$ is the power set of *Entity*. We define $\mathcal{S}_\mathcal{C}$ to be the least function[4] $\mathsf{rmem} : \mathit{Role} \to \wp(\mathit{Entity})$ that satisfies the following system of set inequalities:

$$\{\ \mathsf{rmem}(A.r) \supseteq \mathsf{expr}[\mathsf{rmem}](e) \mid A.r \longleftarrow e \in \mathcal{C}\ \}$$

where $\mathsf{expr}[\mathsf{rmem}] : \mathit{RoleExpression} \to \wp(\mathit{Entity})$ is defined as follows:

$$\begin{array}{rcl} \mathsf{expr}[\mathsf{rmem}](B) & = & \{B\} \\ \mathsf{expr}[\mathsf{rmem}](A.r) & = & \mathsf{rmem}(A.r) \\ \mathsf{expr}[\mathsf{rmem}](A.r_1.r_2) & = & \bigcup_{B \in \mathsf{rmem}(A.r_1)} \mathsf{rmem}(B.r_2) \\ \mathsf{expr}[\mathsf{rmem}](f_1 \cap \cdots \cap f_k) & = & \bigcap_{1 \le j \le k} \mathsf{expr}[\mathsf{rmem}](f_j) \end{array}$$

Note that expr naturally extends $\mathcal{S}_\mathcal{C}$ to role expressions: $\mathsf{expr}[\mathcal{S}_\mathcal{C}](e)$ gives the set of members of any role expression e. ∎

We use this least-solution definition for $\mathcal{S}_\mathcal{C}$ because credentials may define roles recursively. In the following paragraphs, we show that $\mathcal{S}_\mathcal{C}$ is well defined and present a straightforward way to construct the least solution finitely.

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

We now define the function rmem^∞, which maps each role in $\mathsf{Roles}(\mathcal{C})$ to a set of entities in $\mathsf{Entities}(\mathcal{C})$, to be the limit of a sequence $\{\mathsf{rmem}^i\}_{i \in \mathcal{N}}$, where $\mathcal{N}$ is the set of natural numbers, and where for each i, $\mathsf{rmem}^i : \mathsf{Roles}(\mathcal{C}) \to \wp(\mathsf{Entities}(\mathcal{C}))$. This sequence of functions is defined inductively by taking $\mathsf{rmem}^0(A.r) = \emptyset$ for each role $A.r$ in $\mathsf{Roles}(\mathcal{C})$, and by defining $\mathsf{rmem}^{i+1} = \mathsf{iterate}[\mathsf{rmem}^i]$, where

$$\mathsf{iterate}[\mathsf{rmem}](A.r) = \bigcup_{A.r \longleftarrow e \,\in\, \mathcal{C}} \mathsf{expr}[\mathsf{rmem}](e)$$

The set of all functions $f : \mathsf{Roles}(\mathcal{C}) \to \wp(\mathsf{Entities}(\mathcal{C}))$ forms a finite lattice, and $\mathsf{iterate}$ is a monotonic function over this lattice, because the operators used to construct it ($\cap$ and $\cup$) are monotonic. Therefore, the least fixpoint of $\mathsf{iterate}$ is known to exist. Consider the function $g : \mathit{RoleExpression} \to \wp(\mathit{Entity})$ that maps any role $A.r$ in $\mathsf{Roles}(\mathcal{C})$ to $\mathsf{rmem}^\infty(A.r)$ and any role not in $\mathsf{Roles}(\mathcal{C})$ to

[3]In this paper, we always assume that the set of credentials is finite.

[4]One function $f : \mathit{Role} \to \wp(\mathit{Entity})$ is less than another $g : \mathit{Role} \to \wp(\mathit{Entity})$ if $f(A.r) \subseteq g(A.r)$ for every role $A.r \in \mathit{Role}$.

the empty set. The function g is the semantic function $\mathcal{S_C}$: it clearly satisfies the system of set inequalities in Definition 1, and one can show by induction that any function that satisfies these inequalities is greater than or equal to g.

Example 3 We revise the scenario in Example 2 slightly, reusing and renumbering several of those credentials. Now EPub offers a special discount to anyone who is both a preferred customer of EOrg and an ACM member, and Alice is both.

$$
\begin{array}{llllr}
\text{EPub.spdiscount} \longleftarrow \text{EOrg.preferred} \cap \text{ACM.member} & & & & (1) \\
\text{EOrg.preferred} \longleftarrow \text{EOrg.university.student} & & & & (2) \\
\text{EOrg.university} \longleftarrow \text{ABU.accredited} & (3) & \text{ABU.accredited} \longleftarrow \text{StateU} & & (4) \\
\text{StateU.student} \longleftarrow \text{RegistrarB.student} & (5) & \text{RegistrarB.student} \longleftarrow \text{Alice} & & (6) \\
\text{ACM.member} \longleftarrow \text{Alice} & (7) & & &
\end{array}
$$

We now give the computation of the semantics of this set of credentials. Showing only the new elements added to the function's value in each step, the successive values of rmem^i are as follows:

for $i = 1$, ABU.accredited $\mapsto$ {StateU}, RegistrarB.student $\mapsto$ {Alice}, ACM.member $\mapsto$ {Alice}
for $i = 2$, EOrg.university $\mapsto$ {StateU}, StateU.student $\mapsto$ {Alice}
for $i = 3$, EOrg.preferred $\mapsto$ {Alice}
for $i = 4$, EPub.spdiscount $\mapsto$ {Alice}

The sequence stabilizes at $i = 4$. ∎

3 Centralized Credential Chain Discovery

In this section, we study credential chain discovery for RT_0, under the assumption that all the relevant credentials are stored centrally. Given a set $\mathcal{C}$ of RT_0 credentials, three common kinds of queries (goals) are:

1. Given a role expression e and an entity D, determine whether $D \in \mathsf{expr}[\mathcal{S_C}](e)$.
2. Given a role expression e, determine its member set, $\mathsf{expr}[\mathcal{S_C}](e)$.
3. Given an entity D, determine all the roles it is a member of, *i.e.*, all role $A.r$'s such that $D \in \mathcal{S_C}(A.r)$.

When an entity D submits a request to access a particular resource, and it is the access mediator's policy to authorize members of the role $A.r$ to access the resource, then the access mediator needs to determine whether $D \in \mathcal{S_C}(A.r)$. We call this a query (or a goal). Recall that $\mathsf{expr}[\mathcal{S_C}](A.r) = \mathcal{S_C}(A.r)$, and so $D \in \mathcal{S_C}(A.r)$ is a query of the first kind. The other two kinds are useful for evaluating the effects of credentials. They are used for determining all entities that are granted a certain permission and all permissions granted to a particular entity. They are also used in the course of answering the first kind of query, in order to handle linked roles. In this section, we give goal-directed algorithms for answering these three kinds of queries.

3.1 Algorithm Requirements and Related Work

Chain discovery in RT_0 shares two key characteristics with discovery in SDSI: linked names give credential chains a non-linear structure and role definitions can be recursive. Cyclic dependencies must be handled carefully, to avoid non-termination. Clarke *et al.* [6] have given an algorithm for chain discovery in SPKI/SDSI 2.0. Their algorithm views each credential as a rewriting rule and views discovery as a term-rewriting problem. The algorithm handles cyclic dependency by using a bottom-up approach, performing a closure operation over the set of all credentials before it finds one chain. This may be suitable when large numbers of queries are made about a slowly changing credential pool of modest size. However, when the credential pool is large, or when the frequency of changes to the credential pool (particularly deletions, such as credential expirations or revocations) approaches the frequency of queries against the pool, the efficiency of the bottom-up approach deteriorates rapidly.

Li [15] gave a 4-rule logic program to calculate meanings of SDSI credentials. Cyclic dependencies are handled by using XSB [11] to evaluate the program. XSB, unlike most other Prolog engines, uses an extension table mechanism and guarantees termination of a large class of programs, including, but not limited to, all Datalog programs. Yet, for many trust-management applications, this solution is excessively heavy-weight. Even the stripped down version of XSB is several megabytes, while the jar file of the current RT_0 engine is less than 40KB. Moreover, based on our past experience using XSB [14, 15], it is often hard to integrate with XSB closely; and one has less control than needed during the inference process. For example, it would be hard to interleave credential collection with inferencing steps, as needed for distributed chain discovery.

Because we seek techniques that work well when the credential pool is distributed or changes frequently, we require chain discovery algorithms that are goal-directed and that can drive the collection process. They also must support interleaving credential collection and chain construction (*i.e.*, evaluation) steps.

We meet these requirements by providing graph-based algorithms. Credentials are represented by edges. Entities, roles, and other role expressions are represented by nodes. Chain discovery is performed by starting at the node representing the requester, or the node representing the role (permission) to be proven, or both, and then traversing paths in the graph trying to build an appropriate chain. In addition to being goal-directed, this approach allows the elaboration of the graph to be scheduled flexibly. Also, the graphical representation of the evaluation state makes it relatively straightforward to manage cyclic dependencies. Graph-based approaches to chain discovery have been used before, *e.g.*, by Aura [1] for SPKI delegation certificates and by Clarke *et al.* [6] for SDSI name certificates without linked names. However, neither of them deals with linked names. To our knowledge, our algorithms are the first to use a graphical representation to handle linked roles.

3.2 Credential Graphs

A *credential graph* is a directed graph that represents a set $\mathcal{C}$ of credentials and the meanings of these credentials. Each node in a credential graph represents a role expression. Every credential $A.r \longleftarrow e$ in $\mathcal{C}$ (no matter what type of credential it is) contributes an edge[5] $A.r \leftarrow e$, called a credential edge. Additional edges are added to handle linked roles and intersections. We call these *derived* edges because their inclusion comes from the existence of other, semantically related, paths in the graph. Each derived edge has one or more *support sets*; each support set is a set of one or more paths that together justify the existence of the derived edge.

Given a set $\mathcal{C}$ of credentials, recall that $\mathsf{Entities}(\mathcal{C})$ is the set of entities in $\mathcal{C}$, $\mathsf{Names}(\mathcal{C})$ is the

[5] In this paper, long arrows ($\longleftarrow$) represent credentials, short arrows ($\leftarrow$) represent edges, and short arrows with stars ($\overset{*}{\leftarrow}$) represent paths, which consist of zero or more edges.

set of role names in $\mathcal{C}$. We define the following: $\mathsf{Intersections}(\mathcal{C})$ is the set of intersections in $\mathcal{C}$, and $\mathsf{FExps}(\mathcal{C})$ is the set of intersection-free role expressions that can be constructed using $\mathsf{Entities}(\mathcal{C})$ and $\mathsf{Names}(\mathcal{C})$, *i.e.*, $\mathsf{FExps}(\mathcal{C}) = \{A,\ A.r_1,\ A.r_1.r_2 \mid A \in \mathsf{Entities}(\mathcal{C}),\ r_1, r_2 \in \mathsf{Names}(\mathcal{C})\}$.

Definition 2 (Credential Graphs) Each credential graph for $\mathcal{C}$ is parameterized by a finite set of role expressions $Q \subseteq \mathit{RoleExpression}$. (Recall that *RoleExpression* is the set of all role expressions in RT_0.) The credential graph $G_{\mathcal{C}:Q}$ has node set $N_{\mathcal{C}:Q}$ and edge set $E_{\mathcal{C}:Q}$, defined as follows:

$$N_{\mathcal{C}:Q} = \mathsf{FExps}(\mathcal{C}) \cup \mathsf{Intersections}(\mathcal{C}) \cup Q$$

$E_{\mathcal{C}:Q}$ is the least set of edges over $N_{\mathcal{C}:Q}$ that satisfies the following three closure properties:

Closure Property 1: If $A.r \longleftarrow e \in \mathcal{C}$, then $A.r \leftarrow e \in E_{\mathcal{C}:Q}$. We call $A.r \leftarrow e$ a *credential edge.*

Closure Property 2: If there is a path $A.r_1 \stackrel{*}{\leftarrow} B$ in $G_{\mathcal{C}:Q}$, then $A.r_1.r_2 \leftarrow B.r_2 \in E_{\mathcal{C}:Q}$. We call $A.r_1.r_2 \leftarrow B.r_2$ a *derived link edge*, and call the path $A.r_1 \stackrel{*}{\leftarrow} B$ a *support set* for the edge.

Closure Property 3: If D, $f_1 \cap \cdots \cap f_k \in N_{\mathcal{C}:Q}$, and for each $j \in [1..k]$ there is a path $f_j \stackrel{*}{\leftarrow} D$ in $G_{\mathcal{C}:Q}$, then $f_1 \cap \cdots \cap f_k \leftarrow D \in E_{\mathcal{C}:Q}$. We call this a *derived intersection edge*, and say that the union of the paths $\{\, f_j \stackrel{*}{\leftarrow} D \mid j \in [1..k] \,\}$ is a *support set* for the edge.

The definition of $E_{\mathcal{C}:Q}$ can be made effective by inductively constructing a sequence of edge sets $\{E^i_{\mathcal{C}:Q}\}_{i \in \mathcal{N}}$ whose limit is $E_{\mathcal{C}:Q}$. We take $E^0_{\mathcal{C}:Q} = \{A.r \leftarrow e \mid A.r \longleftarrow e \in \mathcal{C}\}$ and construct $E^{i+1}_{\mathcal{C}:Q}$ from $E^i_{\mathcal{C}:Q}$ by adding one edge according to either closure property 2 or 3. Since $N_{\mathcal{C}:Q}$ is finite, we do not have to worry about scheduling these additions. At some finite stage, no more edges will be added, and the sequence converges to $E_{\mathcal{C}:Q}$.

Intuitively, Q is a set of role expressions that do not appear in $\mathsf{FExps}(\mathcal{C})$ or $\mathsf{Intersections}(\mathcal{C})$, but that nonetheless hold interest, *e.g.*, intersections that can be constructed using role expressions in $\mathsf{FExps}(\mathcal{C})$, but do not appear in $\mathcal{C}$. When $Q = \emptyset$, we call $G_{\mathcal{C}:\emptyset}$ the *basic credential graph* of $\mathcal{C}$, and use $G_{\mathcal{C}}$ to denote this graph, $N_{\mathcal{C}}$ to denote its node set, and $E_{\mathcal{C}}$ to denote its edge set. $G_{\mathcal{C}:Q}$ is always a supergraph of $G_{\mathcal{C}}$, and when $Q \subseteq \mathsf{FExps}(\mathcal{C}) \cup \mathsf{Intersections}(\mathcal{C})$, $G_{\mathcal{C}:Q} = G_{\mathcal{C}}$. ∎

Given a set $\mathcal{C}$ of credentials, we define the size of $\mathcal{C}$ to be $\mathsf{size}(\mathcal{C}) = \sum_{A.r \longleftarrow e \in \mathcal{C}} |e|$, where $|A| = |A.r| = |A.r_1.r_2| = 1$ and $|f_1 \cap \cdots \cap f_k| = k$.

Let N be the number of credentials in $\mathcal{C}$ and M be $\mathsf{size}(\mathcal{C})$. The basic credential graph, $G_{\mathcal{C}}$, of $\mathcal{C}$, has N credential edges. We now show that $G_{\mathcal{C}}$ has $O(N^2M)$ derived link edges and $O(N^2)$ derived intersection edges. To add $A.r_1.r_2 \leftarrow B.r_2$, the role node $A.r_1$ must have an incoming edge and the entity node B must have an outgoing edge. If a role node has an incoming edge, then the edge must be a credential edge; there are $O(N)$ such roles. If an entity node has outgoing edges, then at least one such edge is a credential edge; there are $O(N)$ such entities. Furthermore, there are $O(M)$ possible r_2's; therefore, there are $O(N^2M)$ derived link edges. Moreover, because there are $O(N)$ intersections in $G_{\mathcal{C}}$, there are $O(N^2)$ derived intersection edges. Thus, there are $O(N^2M)$ edges in $G_{\mathcal{C}}$.

Figure 1 gives part of the basic credential graph of the credentials in Example 3.

Theorem 1 (Soundness of Credential Graphs) *Given a set $\mathcal{C}$ of credentials and two role expressions e_1 and e_2, if there is a path $e_2 \stackrel{*}{\leftarrow} e_1$ in any $G_{\mathcal{C}:Q}$, then $\mathsf{expr}[\mathcal{S}_{\mathcal{C}}](e_2) \supseteq \mathsf{expr}[\mathcal{S}_{\mathcal{C}}](e_1)$. In particular, if there is a path $e \stackrel{*}{\leftarrow} D$ in $G_{\mathcal{C}:Q}$, then $D \in \mathsf{expr}[\mathcal{S}_{\mathcal{C}}](e)$.*

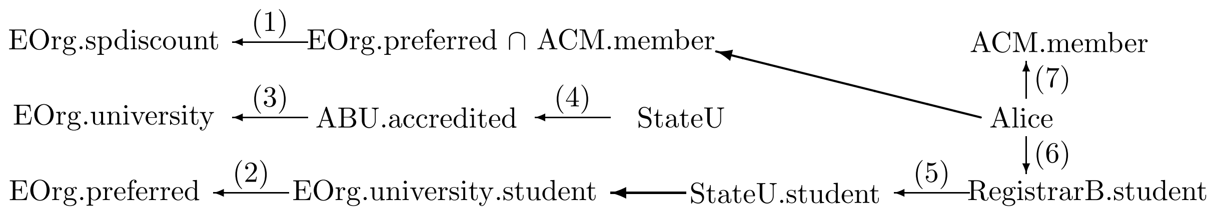


Figure 1: A subgraph of the basic credential graph for the set of credentials in Example 3. (Some nodes and derived linked edges are not shown.) Edges with numbers are credential edges; and the numbers correspond to the numbers in Example 3. The two edges without numbers are derived edges; one added by closure property 2, and one by closure property 3.

Proof. The proof is by induction on the steps of the construction of $\{E^i_{\mathcal{C}:Q}\}_{i\in\mathcal{N}}$ in Definition 2.

We show the base case by using a second, inner induction on the length of the path $e_2 \stackrel{*}{\leftarrow} e_1$ in $E^0_{\mathcal{C}:Q}$. The inner base case, in which $e_1 = e_2$, is trivial; we consider the step. We decompose $e_2 \stackrel{*}{\leftarrow} e_1$ into $e_2 \leftarrow e' \stackrel{*}{\leftarrow} e_1$. Because each edge in $E^0_{\mathcal{C}:Q}$ corresponds to a credential, we have $e_2 \longleftarrow e' \in \mathcal{C}$. It follows that $\mathsf{expr}[\mathcal{S}_\mathcal{C}](e_2) \supseteq \mathsf{expr}[\mathcal{S}_\mathcal{C}](e')$, by definition of $\mathcal{S}_\mathcal{C}$. The induction hypothesis gives us $\mathsf{expr}[\mathcal{S}_\mathcal{C}](e') \supseteq \mathsf{expr}[\mathcal{S}_\mathcal{C}](e_1)$, so $\mathsf{expr}[\mathcal{S}_\mathcal{C}](e_2) \supseteq \mathsf{expr}[\mathcal{S}_\mathcal{C}](e_1)$.

We prove the step by again using an inner induction on the length of $e_2 \stackrel{*}{\leftarrow} e_1$, which we now assume is in $E^{i+1}_{\mathcal{C}:Q}$. Again the basis is trivial. For the step, we decompose $e_2 \stackrel{*}{\leftarrow} e_1$ into $e_2 \leftarrow e' \stackrel{*}{\leftarrow} e_1$. There are three cases for the edge $e_2 \leftarrow e'$.

case 1: When $e_2 \leftarrow e'$ is a credential edge, the argument proceeds along the same lines as the base case, using the inner induction hypothesis to derive $\mathsf{expr}[\mathcal{S}_\mathcal{C}](e') \supseteq \mathsf{expr}[\mathcal{S}_\mathcal{C}](e_1)$.

case 2: When $e_2 \leftarrow e'$ is a derived link edge, e_2 has the form $A.r_1.r_2$, e' has the form $B.r_2$, and there is a path $A.r_1 \stackrel{*}{\leftarrow} B$ in $E^i_{\mathcal{C}:Q}$. The outer induction hypothesis gives us $\mathsf{expr}[\mathcal{S}_\mathcal{C}](A.r_1) \supseteq \mathsf{expr}[\mathcal{S}_\mathcal{C}](B)$, *i.e.*, $B \in \mathcal{S}_\mathcal{C}(A.r_1)$. The inner induction hypothesis gives us $\mathsf{expr}[\mathcal{S}_\mathcal{C}](B.r_2) \supseteq \mathsf{expr}[\mathcal{S}_\mathcal{C}](e_1)$. Together with the definition of expr for $A.r_1.r_2$, these imply $\mathsf{expr}[\mathcal{S}_\mathcal{C}](e_2) \supseteq \mathsf{expr}[\mathcal{S}_\mathcal{C}](e_1)$, as required.

case 3: When $e_2 \leftarrow e'$ is a derived intersection edge, e_2 has the form $f_1 \cap \cdots \cap f_k$, $e' = e_1$ is an entity D (because entity nodes have no incoming edges), and there are paths $f_j \stackrel{*}{\leftarrow} D$ in $E^i_{\mathcal{C}:Q}$ for each $j \in [1..k]$. The outer induction hypothesis gives us $D \in \mathsf{expr}[\mathcal{S}_\mathcal{C}](f_j)$ for each $j \in [1..k]$; therefore, $\mathsf{expr}[\mathcal{S}_\mathcal{C}](e_2) \supseteq \mathsf{expr}[\mathcal{S}_\mathcal{C}](e_1)$. ∎

Theorem 2 (Completeness of Credential Graphs) *Given a set $\mathcal{C}$ of credentials, an entity D, and a role expression e_0 such that $D \in \mathsf{expr}[\mathcal{S}_\mathcal{C}](e_0)$, then there exists $e_0 \stackrel{*}{\leftarrow} D$ in $G_{\mathcal{C}:\{e_0\}}$. In particular, when $e_0 \in \mathsf{FExps}(\mathcal{C}) \cup \mathsf{Intersections}(\mathcal{C})$, there exists $e_0 \stackrel{*}{\leftarrow} D$ in $G_\mathcal{C}$.*

See Appendix A.1 for the proof.

3.3 Overview of Credential-graph-based Search Algorithms

Together, Theorems 1 and 2 tell us that, given a set $\mathcal{C}$ of credentials, we can answer each of the queries enumerated at the top of this section by consulting a credential graph of $\mathcal{C}$. Constructing the path $e \stackrel{*}{\leftarrow} D$ alone proves $D \in \mathsf{expr}[\mathcal{S}_\mathcal{C}](e)$. However, where $e \stackrel{*}{\leftarrow} D$ contains derived edges, each derived edge must have at least one support set for it constructed first. The portion of the credential graph that must be constructed is what we call a credential chain.

Definition 3 (Credential Chains) Given a set $\mathcal{C}$ of credentials and two role expressions e_1 and e_2, a *credential chain* from e_1 to e_2, denoted $\langle e_2 \twoheadleftarrow e_1\rangle$, is a minimal subset of $E_{\mathcal{C}:\{e_2\}}$ containing a path $e_2 \xleftarrow{*} e_1$ and also containing a support set for each derived edge in the subset. We use $\mathcal{C}\langle e_2 \twoheadleftarrow e_1\rangle$ to denote the set of credentials in $\langle e_2 \twoheadleftarrow e_1\rangle$, *i.e.*, the set of credentials corresponding to credential edges in $\langle e_2 \twoheadleftarrow e_1\rangle$.

If a chain $\langle e_2 \twoheadleftarrow e_1\rangle$ consists entirely of credential edges, then $\langle e_2 \twoheadleftarrow e_1\rangle$ is a linear path $e_2 \xleftarrow{*} e_1$. And if $\langle e_2 \twoheadleftarrow e_1\rangle$ has any derived edges, then the path $e_2 \xleftarrow{*} e_1$ must contain derived edges. When $e_2 \xleftarrow{*} e_1$ contains a derived link edge $A.r_1.r_2 \leftarrow B.r_2$, $e_2 \xleftarrow{*} e_1$ can be decomposed as $e_2 \xleftarrow{*} A.r_1.r_2 \leftarrow B.r_2 \xleftarrow{*} e_1$, and $\langle e_2 \twoheadleftarrow e_1\rangle = \langle e_2 \twoheadleftarrow A.r_1.r_2\rangle \cup \langle A.r_1 \twoheadleftarrow B\rangle \cup \{A.r_1.r_2 \leftarrow B.r_2\} \cup \langle B.r_2 \twoheadleftarrow e_1\rangle$. When $e_2 \xleftarrow{*} e_1$ contains a derived intersection edge $f_1 \cap \cdots \cap f_k \leftarrow D$, $e_2 \xleftarrow{*} e_1$ can be decomposed as $e_2 \xleftarrow{*} f_1 \cap \cdots \cap f_k \leftarrow D$, and $\langle e_2 \twoheadleftarrow e_1\rangle = \langle e_2 \twoheadleftarrow f_1 \cap \cdots \cap f_k\rangle \cup \langle f_1 \twoheadleftarrow D\rangle \cup \cdots \cup \langle f_k \twoheadleftarrow D\rangle \cup \{f_1 \cap \cdots \cap f_k \leftarrow D\}$. A chain $\langle e_2 \twoheadleftarrow e_1\rangle$ is always a subset of $E_{\mathcal{C}\langle e_2 \twoheadleftarrow e_1\rangle:\{e_2\}}$.

The rest of this section gives algorithms that can answer the three kinds of queries, listed at the top of this section, without constructing a complete credential graph. The backward search algorithm, to be presented in Section 3.4, answers the second form of queries, *i.e.*, it determines all members of a role expression. The forward search algorithm in Section 3.5 answers the third form of queries, *i.e.*, it determines all roles that an entity is a member of. The bidirectional search algorithm in Section 3.6 answers the first kind of queries, *i.e.*, it determines whether an entity is a member of a role expression. We then discuss some enhancements of these algorithms in Section 3.7 and show how to apply these algorithms to SDSI 2.0 credentials in Section 3.8.

In Section 4, we will give a distributed search algorithm that subsumes the three algorithms presented in this section. There, we give pseudo-code of the algorithm and prove the correctness of the algorithm. Readers interested in the details of the three algorithms presented in this section may consult Section 4.

3.4 The Backward Search Algorithm

We present a backward search algorithm that can determine all the members of a given role expression e_0. In terms of the credential graph, it finds all the entity nodes that can reach the node e_0, and for each such entity D, it constructs every chain $\langle e_0 \twoheadleftarrow D\rangle$. We call it backward because it follows edges in the reverse direction. This algorithm works by constructing a *proof graph*, which is a data structure that represents a credential graph and maintains certain information on the nodes.

Algorithm 1 (Backward Search: $\mathsf{backward}(e_0)$) The backward search algorithm constructs a proof graph, maintaining a queue of nodes to be processed; both initially contain just one node, e_0. Nodes are processed one by one until the queue is empty.

In the proof graph, there is only one node corresponding to each role expression and each edge is added only once. Each time the algorithm tries to create a node for a role expression e, it first checks whether such a node already exists; if not, it creates a new node, adds it into the queue, and uses it. Otherwise, it uses the existing node.

With each node e, the algorithm stores a set of (backward) solutions, which is the set of entity nodes, D, that can reach e (*i.e.*, $e \xleftarrow{*} D$). Each node e also stores a set of (backward) solution monitors, which is the set of objects that want to be notified when the current node receives a solution. Such objects include nodes that e can reach directly and objects created to handle linked roles and intersections, which will be described below.

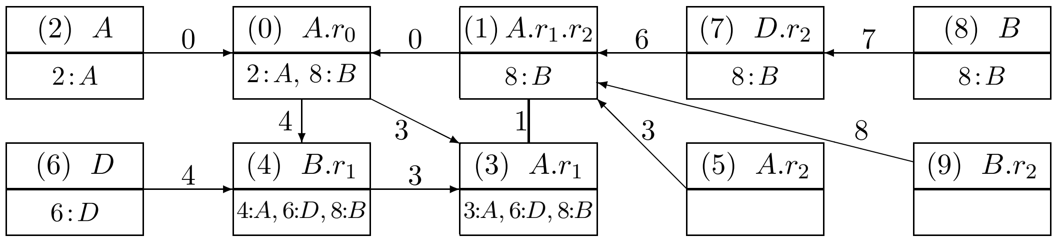


Figure 2: The proof graph constructed by doing backward search from $A.r_0$ with the following set of credentials $\{\, A.r_0 \longleftarrow A.r_1.r_2,\ A.r_0 \longleftarrow A,\ A.r_1 \longleftarrow B.r_1,\ A.r_1 \longleftarrow A.r_0,\ B.r_1 \longleftarrow A.r_0,\ B.r_1 \longleftarrow D,\ D.r_2 \longleftarrow B,\ B.r_0 \longleftarrow A.r_0,\ D.r_1 \longleftarrow D.r_2.r_3 \,\}$. The first line of each node gives the node number in order of creation and the role expression represented by the node. The second line lists each solution eventually associated with this node. Each of those solutions and each graph edge is labeled by the number of the node that was being processed when the solution or edge was added. The edge labelled with 1 is a linking monitor.

Solutions are propagated from e to e's solution monitors as follows. When a node is notified to add a solution, it checks whether the solution exists in its solution set; if not, it adds the solution and then notifies all its solution monitors about this new solution. When a node e_2 is first added as a solution monitor of e_1 (*e.g.*, as the result of adding $e_2 \leftarrow e_1$), all existing solutions on e_1 are copied to e_2.

Nodes are processed as follows:

- To process a role node $A.r$, the algorithm finds all credentials that define $A.r$. For each credential $A.r \longleftarrow e$, it creates a node for e, and adds the credential edge $A.r \leftarrow e$.
- To process an entity node B, the algorithm notifies the node to add B as a solution to itself.
- To process a linked role node $A.r_1.r_2$, the algorithm creates a node for $A.r_1$ and creates a *(backward) linking monitor* and adds it to $A.r_1$'s solution monitors. The monitor, on observing that $A.r_1$ has received a new solution B, creates a node for $B.r_2$ and adds the derived link edge $A.r_1.r_2 \leftarrow B.r_2$.
- To process an intersection node $e = f_1 \cap \cdots \cap f_k$, the algorithm creates one *intersection monitor*, for e, and k nodes, one for each f_j. It then adds the monitor to the solution monitors of each node f_j. This monitor counts how many times it observes that an entity D is added. If the count reaches k, then the monitor adds the derived intersection edge $e \leftarrow D$.

To summarize, in addition to the nodes and edges in the credential graph, the algorithm constructs monitors that implement closure properties 2 and 3. See Section 4 for more details. ∎

The backward search algorithm uses credentials in a demand driven way. It uses credentials only when processing role nodes; each such processing step finds credentials defining a specific role $A.r$ and creates credential edges for them. Figure 2 shows an example of using the backward search algorithm; the goal there is to find all members of the role $A.r_0$.

We next consider the worst-case complexity of backward search. Recall that given a set $\mathcal{C}$ of credentials, $\mathsf{size}(\mathcal{C}) = \sum_{A.r \longleftarrow e \in \mathcal{C}} |e|$, where $|A| = |A.r| = |A.r_1.r_2| = 1$ and $|f_1 \cap \cdots \cap f_k| = k$.

Theorem 3 *Given a set $\mathcal{C}$ of credentials, let N be the number of credentials in $\mathcal{C}$, and $M = \mathsf{size}(\mathcal{C})$. Assuming that finding all credentials that define a role takes time linear in the number of such* *credentials (e.g., by using hashing), the worst-case time complexity of the backward search algorithm is $O(N^3+NM)$ (which is $O(N^3)$ when the size of each intersection in $\mathcal{C}$ is bounded by $O(N)$), and the worst-case space complexity is $O(NM)$.*

The proof analyzes the structure of the proof graph constructed by the algorithm. See Appendix A.2. To see that $O(N^3)$ is a tight bound for the algorithm, consider the following example:

$$\mathcal{C} = \left\{ \begin{array}{c} A_0.r_0 \longleftarrow A_i,\ A_0.r_i \longleftarrow A_0.r_{i-1 \bmod n}, \\ A_i.r_0 \longleftarrow A_{i-1 \bmod n}.r_0,\ A_0.r' \longleftarrow A_0.r_i.r_0 \end{array} \middle| \ 0 \le i < n \right\}$$

There are $N = 4n$ credentials. Doing backward search from $A_0.r'$ constructs n^2 edges of the form $A_0.r_i.r_0 \leftarrow A_j.r_0$, where $0 \le i, j < n$. Each $A_j.r_0$ gets n solutions, so the time complexity is n^3. We can see that intersections are not required to achieve the worst-case time complexity of this algorithm. $O(NM)$ is a tight space bound. The following is an example that reaches the bound:

$$\mathcal{C} = \left\{ \begin{array}{c} A_0.r_0 \longleftarrow A_i,\ A_0.r_i \longleftarrow A_0.r_{i-1 \bmod n}, \\ A_0.r' \longleftarrow A_0.r_i.r_0 \cap A_0.r_i.r_1 \cap \cdots \cap A_0.r_i.r_{K-1} \end{array} \middle| \ 0 \le i < n\} \right\}$$

Note that the worst-case complexity is not the whole picture here, for the following reasons. First, only the parts of credential graphs that are relevant for answering the query are constructed. For example, in Figure 2, credentials $B.r_0 \longleftarrow A.r_0, D.r_1 \longleftarrow D.r_2.r_3$ are not used. Second, the worst case complexities are only reached by pathological examples. To reach the $O(N^3)$ time complexity bound, a set of credentials needs to satisfy three conditions: First, it has $O(N)$ linked roles. Second, a large number ($O(N)$) of linked roles need to be expensive, in which a linked role $A.r_1.r_2$ is expensive if $A.r_1$ has $O(N)$ solutions. Third, for each such expensive linked role, a large number ($O(N)$) of solutions need to be expensive, in which a solution B is expensive if $B.r_2$ has $O(N)$ solutions. Policies in practice are unlikely to satisfy all these conditions. Consider the following example:

Example 4 We extend the scenario in Example 3. In addition to the credentials there, we also have thousands of universities that are certified by ABU and millions of students certified by universities. In addition we have all ACM membership credentials and all IEEE membership credentials. ∎

In Example 4, a backward search from EPub.spdiscount takes time linear in the total number of student credentials and ACM membership credentials. So the complexity is $O(N)$ rather than $O(N^3)$. However, even linear complexity seems quite unacceptable if we just want to know whether Alice is entitled to the discount. The forward search algorithm in the next section addresses this issue.

3.5 The Forward Search Algorithm

The forward search algorithm answers queries of the third form listed at the top of this section, *i.e.*, it finds all roles that contain a given entity as a member. The direction of the search moves from the subject of a credential towards its issuer.

Algorithm 2 (Forward Search: $\mathsf{forward}(D_0)$) The forward algorithm has the same overall structure as the backward algorithm. It constructs a proof graph, maintaining a queue of nodes to be processed; both initially contain just one node, D_0. Nodes are processed one by one until the queue is empty.

On each node e, the algorithm stores a set of (forward) solutions. There are two kinds of solutions: full solutions and partial solutions. Each *full solution* on e is a role node that is reachable from e. Each *partial solution* has the form $(f_1 \cap \cdots \cap f_k,\ j)$, where $1 \leq j \leq k$. The node e gets the solution $(f_1 \cap \cdots \cap f_k,\ j)$ when f_j is reachable from e. Such a partial solution is just one piece of a proof that e can reach $f_1 \cap \cdots \cap f_k$. It is passed through edges in the same way as is a full solution. When an entity node D gets the partial solution, it checks whether it has all k pieces; if it does, it adds the edge $f_1 \cap \cdots \cap f_k \leftarrow D$.

Each node e also stores a set of (forward) solution monitors, which includes the nodes that can reach e directly, as well as objects created to handle linked roles. The addition of an edge $e_2 \leftarrow e_1$ results in e_1 being added as a solution monitor on e_2.

Forward processing a node corresponding to a role expression e has the following three steps.

1. If e is a role $B.r_2$, add the role as a solution to the node itself, then create a (forward) linking monitor and add it as a solution monitor on B. This monitor, on observing that B gets a full solution $A.r_1$, creates the node $A.r_1.r_2$ and adds the edge $A.r_1.r_2 \leftarrow B.r_2$.
2. Find all credentials of the form $A.r \longleftarrow e$; for each such credential, create a node for $A.r$, if none exists, and add the edge $A.r \leftarrow e$.
3. If e is not an intersection, find all credentials of the form $A.r \longleftarrow f_1 \cap \cdots \cap f_k$ such that some $f_j = e$; then add $(f_1 \cap \cdots \cap f_k,\ j)$ as a partial solution on e. ∎

The forward search algorithm uses credentials in a demand driven way. Each node-processing step finds every credential that has a specific role expression e as its body and also every credential that uses e as part of its body (when the body is an intersection). Figure 3 shows the result of doing forward search using a subset of the credentials in Examples 2 and 3. The goal there is to find all the roles that Alice is a member of.

Theorem 4 *Assuming that finding all credentials that use a specific role expression e in their bodies takes time linear in the number of such credentials, the worst-case time complexity for the forward search algorithm is $O(N^2M)$, and the worst-case space complexity is $O(NM)$.*

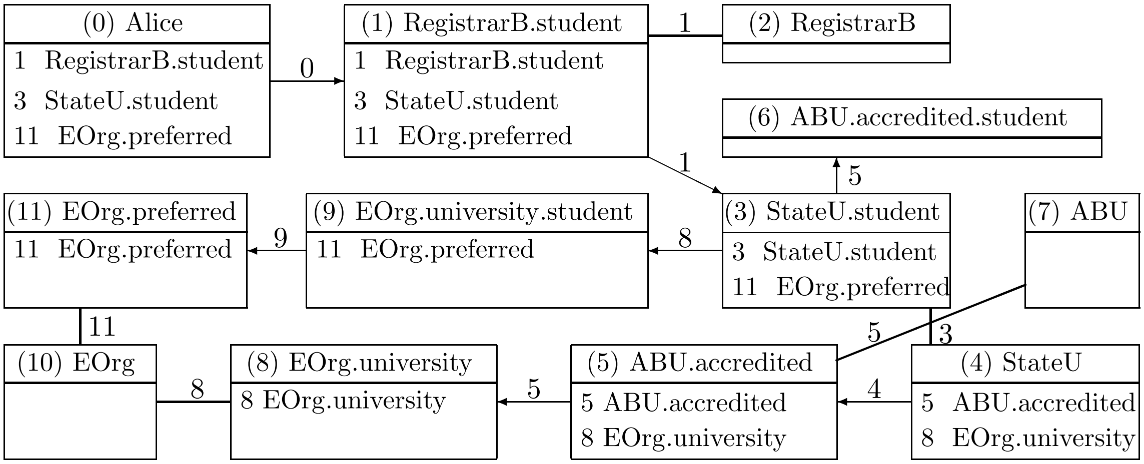
See Appendix A.3 for the proof and an example that reaches the worst-case complexity bound.

As in the case of the backward search algorithm, the worst-case complexity is only part of the picture here. If we do a forward search from Alice with the credentials in Example 4, the millions of credentials about other students will not be touched. This illustrates the power of goal-directed algorithms and the importance of searching in the right direction.

3.6 The Bidirectional Search Algorithm

To answer queries of the first form listed at the top of this section (*i.e.*, whether a given entity, D, is a member of a given role expression, e) we can use a backward search from e; and when e is a role, we can use a forward search from D as well. We also have the alternative of searching from both D and e at once. We now present such a bidirectional search algorithm.

Algorithm 3 (Bidirectional Search: $\mathsf{bidirectional}(e, D)$**)** This algorithm combines the forward algorithm and the backward algorithm in a straightforward way. It maintains two queues, one forward processing queue and one backward processing queue, and iteratively removes a node form the queue and process it until both queues are empty. Each node e stores backward solutions



$$\left\{ \begin{array}{c} \text{EOrg.preferred} \longleftarrow \text{EOrg.university.student},\ \text{EOrg.university} \longleftarrow \text{ABU.accredited} \\ \text{ABU.accredited} \longleftarrow \text{StateU},\ \text{StateU.student} \longleftarrow \text{RegistrarB.student},\ \text{RegistrarB.student} \longleftarrow \text{Alice} \end{array} \right\}$$

Figure 3: The proof graph constructed by forward search from Alice with the set of credentials listed above. The first line of each node gives the node number in order of creation and the role expression represented by the node. The second part of a node lists each solution eventually associated with this node. Each of those solutions and each graph edge is labelled by the number of the node that was being processed when the solution or edge was added. There are five edges that do not have arrows; they represent linking monitors.

(entities that are members of e), full forward solutions (roles that e is a member of), and partial (forward) solutions resulting from intersections, as well as backward and forward solution monitors. The algorithm carries out a backward search from e and a forward search from D at the same time. Whenever an edge is added, both forward and backward solutions propagate. One knows that there is a chain $\langle e \twoheadleftarrow D \rangle$ when e receives D as a backward solution. When e is a role $A.r$, one can also check whether D receives $A.r$ as a forward solution. One knows that such a chain does not exist if one cannot find these solutions when the search completes. ∎

The above bidirectional search algorithm may construct a larger graph than does either forward or backward search. However, when credential storage is distributed, bidirectional search can succeed even when both backward and forward search fail. The above algorithm is the foundation of Algorithm 4, the distributed search algorithm to be presented in Section 4.

The bidirectional search algorithm can be improved to avoid full backward search or full forward search when they are unnecessary. To answer a query whether $D \in \mathsf{expr}[\mathcal{S_C}](A.r)$, a type-1 credential $A.r \longleftarrow B$ does not need to be used immediately, as it does not contribute to solving the query. One way to improve the algorithm is to pass queries as well as solutions between nodes and to mark nodes to indicate whether backward and/or forward solutions are wanted at them. To determine whether $D \in \mathsf{expr}[\mathcal{S_C}](A.r)$, the improved algorithm adds the query to nodes D and $A.r$. Then, while backward processing a role node $A.r$, unless $A.r$ is marked as wanting all backward solutions, the improved algorithm only looks for credentials of the form $A.r \longleftarrow D$ or of the form $A.r \longleftarrow e$ in which e is not an entity. When a node receives the same query from both directions, we know that the query is true. While backward processing a linked role $A.r_1.r_2$, $A.r_1$ is marked as wanting all backward solutions, which requires normal (unimproved) backward processing on $A.r_1$.

In this paper, we choose not to present the improved algorithm, for the following reasons. First, that algorithm is much more complicated and its correctness proof would be much longer than the one in Section 4. Second, although we expect this improved algorithm to work much better in practice, it does not improve the worst-case complexity. Our purpose in this paper is not to refine a search algorithm. Third, the storage type system to be presented in Section 5 can help improve search efficiency by stopping search in one direction to avoid large fan-out; that makes the improvement here not so crucial.

3.7 Enhancements to Search Algorithms

The three algorithms in this section can be enhanced so that when they answer affirmatively one of the queries listed at the top of this section, they construct and return a chain proving the queried relationship, and/or the credentials from which that chain is derived. For this, each solution stored at a node is associated with the adjacent node from which it was propagated. Using this information, it is straightforward to extract a chain from the proof graph, and to compute the set of credentials being used in the chain. For example, to discover a chain proving that $\mathcal{S}_\mathcal{C}(A.r) \ni D$, one can run a backward search to construct a proof graph. Then, if the node $A.r$ receives D as a backward solution, one knows that $D \in \mathcal{S}_\mathcal{C}(A.r)$. One can extract a chain $\langle A.r \twoheadleftarrow D\rangle$ by tracing backward the path that the solution D traversed from D to $A.r$, collecting all credentials corresponding to edges in that path. When encountering derived edges, one must also traverse the supporting paths they are derived from. One can enhance the forward and bidirectional searches similarly.

The three algorithms in this section use queues to organize node processing, resulting in breadth-first search. A variety of search strategies bear consideration, and different algorithms can be developed based on them. If they used stacks, they would perform depth-first search. In general, when there are several nodes that can be explored (from either direction), they can be placed in a priority queue according to some heuristic criteria, *e.g.*, fan-out.

3.8 Application to SDSI 2.0

Our algorithms can be used to do chain discovery in SDSI 2.0. A *SDSI credential* is a type-1 RT_0 credential, a type-2 RT_0 credential, or a credential of the form $A.r \longleftarrow B.r_1.r_2.\cdots.r_k$. Given a set $\mathcal{C}$ of SDSI credentials, we define $\mathit{RTrans}(\mathcal{C})$ to be the set of RT_0 credentials that are equivalent to $\mathcal{C}$. $\mathit{RTrans}(\mathcal{C})$ is obtained from $\mathcal{C}$ by replacing each $A.r \longleftarrow B.r_1.r_2.\cdots.r_k$ in $\mathcal{C}$ with k RT_0 credentials $\{\, A.r \longleftarrow A.r'_{k-1}.r_k,\ A.r'_{k-1} \longleftarrow A.r'_{k-2}.r_{k-1},\ \cdots,\ A.r'_2 \longleftarrow A.r'_1.r_2,\ A.r'_1 \longleftarrow B.r_1\}$, in which the r'_i's are newly introduced role names.

Theorem 5 *Given a set $\mathcal{C}$ of SDSI credentials, the worst-case time complexity of the backward algorithm, applied to Trans($\mathcal{C}$), is $O(N^3L)$, where N is the number of credentials in $\mathcal{C}$, and L is the length of the longest linked role in $\mathcal{C}$.*

See Appendix A.4 for the proof. This $O(N^3L)$ worst-case complexity is the same as that of the algorithm in Clarke *et al.* [6]. However, note that the algorithms in [6] is bottom-up, while the algorithms presented here are goal-directed.

4 The Distributed Credential Chain Discovery Algorithm

In this section, we study chain discovery when credential storage is distributed. In Section 4.1, we extend the algorithms described in Section 3 and present one unified, distributed search algorithm.

Then, we prove in two steps that the distributed search algorithm is correct in a precise sense. In Section 4.2, we prove the first step: the algorithm is sound and complete with respect to the credential graphs of the set of credentials that the algorithm retrieves. In Section 4.3, we introduce notions of traversability to analyze which chains can be discovered when credential storage is distributed, and prove the second step of the correctness argument: when a chain is traversable, the distributed algorithm can retrieve credentials in the chains. (Sections 4.1 and 4.2 give pseudo-code and detailed proofs, which some readers may wish to skip when reading for the first time.)

4.1 The Distributed Chain Discovery Algorithm

The three algorithms given in the previous section can be used when credential storage is not centralized, but distributed among credentials' subjects and issuers; this is because these algorithms use credentials in a demand driven way. The backward search algorithm asks for credentials that define a specific role. The forward search algorithm asks for credentials that have a specific role expression in their bodies. The bidirectional algorithm does both.

When credential storage is distributed, we assume that there is some mechanism to link an entity A to a server that hosts credentials involving A on A's behalf. A credential involves A if it is issued by A or uses the entity A, a role $A.r$, or a linked role $A.r_1.r_2$ in its body. In the following, when we say "go to A", we mean to contact the server acting on behalf of A for the purpose of obtaining credentials involving A.

When credential storage is distributed, bidirectional search can find some chains that cannot be found by either forward or backward search alone. Each credential defining $A.r$ has A as its issuer, but in the absence of the credential, one does not know its subject or subjects. Consequently, one has no information about where to find such credentials other than to go to A. If such credentials are not stored with A, backward search cannot make progress at the node $A.r$. Similarly, to find credentials using a role expression e, one can go only to entities[6] in $base(e)$. When credentials using e in their bodies are not stored with subjects, forward search cannot make progress at the node e. The bidirectional algorithm works to find a chain $\langle e_0 \twoheadleftarrow D_0 \rangle$ by doing forward search from D_0 and backward search from e_0. It may succeed in finding a chain when using only backward or only forward search fails, *e.g.*, the chain involving two credentials $A.r \longleftarrow B.r_1$ and $B.r_1 \longleftarrow D_0$, with the first one stored with its issuer A and the second one with its subject D_0.

We now present a distributed search algorithm that can collect distributed credentials to construct some chains not found by the bidirectional search algorithm in Section 3.6. It unifies backward, forward, and bidirectional search. In the distributed context, we need a unified algorithm because a backward search may benefit from using forward search as well. For example, backward processing a node $f_1 \cap \cdots \cap f_k$ creates $f_1, \ldots, f_k$ and results in each of them being backward processed. When one of $f_1, \ldots, f_k$ receives an entity D, one should start forward searching from D as well, because it may be that some path $f_j \xleftarrow{*} D$ can be found only by a bidirectional search. For similar reasons, a distributed forward search may invoke backward search on intersection nodes.

Algorithm 4 (Distributed Chain Discovery) We describe the algorithm in an object-oriented style. We describe two main classes: ProofGraph and ProofNode, and three helper classes: BLinkingMonitor, BIntersectionMonitor, and FLinkingMonitor. We use pseudo-code in Java-style syntax.

[6] Recall that $base(A) = \{A\}$, $base(A.r) = \{A\}$, $base(A.r_1.r_2) = \{A\}$, $base(f_1 \cap \cdots \cap f_k) = base(f_1) \cup \cdots \cup base(f_k)$, and the entities in $base(e)$ are called the subjects of credentials of the form $A.r \longleftarrow e$.

Credentials and their components are shown in the same font as elsewhere in the paper, while other program variables are in typewriter font.

Instance variables of ProofGraph

- **nodes**: maintains all nodes in the graph. Can be implemented by using a HashMap that maps role expressions to nodes.
- **edges**: maintains all edges in the graph. Should support constant-time existence checking and addition of a new edge. Should also be able to retrieve all edges leaving or entering a node in time linear in the number of such edges.
- **b-proc-queue**: backward processing queue. Nodes waiting to be backward processed.
- **f-proc-queue**: forward processing queue. Nodes waiting to be forward processed.

Instance Variables of ProofNode:

- **b-proc-state**: backward processing state. Takes one of the following three values: unprocessed, to-be-processed, processed. When a node is created, its b-proc-state should be unprocessed. When it needs to be b-processed, it is entered into b-proc-queue and its b-proc-state is set to to-be-processed. When the node is taken out and processed, its b-proc-state is set to processed.
- **f-proc-state**: forward processing state. Similar to b-proc-state. Takes one of the following three values: unprocessed, to-be-processed, processed.
- **b-solutions**: the set of backward solutions (entities).
- **f-solutions**: the set of forward solutions (full solutions, which are roles, and partial solutions).
- **b-sol-monitors**: the set of objects that want to know when the current node gets a new backward solution. These objects include backward linking monitors, backward intersection monitors, and all the nodes that the current node can reach directly through an edge.
- **f-sol-monitors**: the set of objects that want to know when the current node gets a new forward solution. These include forward linking monitors, and all the nodes that can reach the current node directly through an edge.

Methods of ProofGraph

```
ProofGraph: run()
{ while (one of b-proc-queue and f-proc-queue is not empty)
  { if (b-proc-queue nonempty) { n = b-proc-queue.dequeue();  n.b-process(); }
    if (f-proc-queue nonempty) { n = f-proc-queue.dequeue();  n.f-process(); } } }
ProofGraph: addNode(e)
{ if (a node n exists for e) { return n; } else { creates one and return it; } }
ProofGraph: addEdge(e2 ← e1)
{ n1 = addNode(e1);  n2 = addNode(e2);  if (edges.contains(n2←n1))) { return; }
  edges.add(n2←n1);
  n1.add-b-sol-monitor(n2);  if (n2.b-proc-state!=unprocessed) { n1.b-activate(); }
  n2.add-f-sol-monitor(n1);  if (n1.f-proc-state!=unprocessed) { n2.f-activate(); } }
```

Methods of ProofNode and helper classes

```
ProofNode(A.r): b-process()
{ b-proc-state = processed;
  creds = find all credentials defining A.r;
  foreach (credential "A.r ←— e" in creds) { addNode(e); addEdge(A.r ← e); } }
ProofNode(A.r1.r2): b-process()
{ b-proc-state = processed;  n = addNode(A.r1);
  n.add-b-sol-monitor(new BLinkingMonitor(A.r1.r2));  n.b-activate(); }
BLinkingMonitor(A.r1.r2): add-b-solution(B)
```

```
20 { addNode(B.r2);  addEdge(A.r1.r2 ← B.r2); }
21 ProofNode(D): b-process()
22  { b-proc-state = processed;  add-b-solution(D); }
23 ProofNode(f1 ∩ ··· ∩ fk): b-process()
24  { b-proc-state = processed;  m = new BIntersectionMonitor(f1 ∩ ··· ∩ fk);
25    foreach (j in 1..k) { n=addNode(fj);  n.add-b-sol-monitor(m);  n.b-activate(); } }
26 BIntersectionMonitor(f1 ∩ ··· ∩ fk): add-b-solution(B)
27  { if (B is being added for the first time) { n = addNode(B);  n.f-activate(); }
28    else if (B has been added k times) { addEdge(f1 ∩ ··· ∩ fk,  B); } }
29 ProofNode(e): f-process()
30  { f-proc-state = processed;
31    if (e is a role B.r2) { add-f-solution(B.r2);  n = addNode(B);
32                 n.add-f-sol-monitor(new FLinkingMonitor(B.r2));  n.f-activate(); }
33    creds = find all credentials having e as its body;
34    foreach (credential A.r ⟵ e) { addNode(A.r);  addEdge(A.r ← e); }
35    if (e is an intersection) { return; }
36    creds = find all credentials like A.r ⟵ f1 ∩ ··· ∩ fk in which fj = e for some j;
37    foreach (credential A.r ⟵ f1 ∩ ··· ∩ fk in creds)
38    { add-f-solution(⟨f1 ∩ ··· ∩ fk, j⟩);  n = addNode(f1 ∩ ··· ∩ fk);  n.b-activate(); } }
39 FLinkingMonitor(B.r2): add-f-solution(A.r1)
40  { addNode(A.r1.r2);  addEdge(A.r1.r2 ← B.r2); }
41 ProofNode(e): b-activate()
42  { if (b-proc-state != unprocessed) { return; }
43    b-proc-state = to-be-processed;  b-proc-queue.enqueue(this);
44    foreach (n such that edges.contains(this←n)) { n.b-activate(); } }
45 ProofNode(e): f-activate()
46  { if (f-proc-state != unprocessed) { return; }
47    f-proc-state = to-be-processed;  f-proc-queue.enqueue(this);
48    foreach (n such that edges.contains(n←this)) { n.f-activate(); } }
49 ProofNode(e): add-b-solution(s)
50  { if (s exists in b-solutions) { return; }  b-solutions.add(s);
51    foreach (n in b-sol-monitors) { n.add-b-solution(s); } }
52 ProofNode(e): add-f-solution(s)
53  { if (s exists in f-solutions) { return; }  f-solutions.add(s);
54    foreach (n in f-sol-monitors) { n.add-f-solution(s);  } }
55    if (e is an entity D && s is a partial solution (f1 ∩ ··· ∩ fk, j)
56        && all k pieces have arrived)  { addEdge(f1 ∩ ··· ∩ fk ← D); } }
57 ProofNode(e): add-b-sol-monitor(n)
58  { b-sol-monitors.add(n);  foreach (s in b-solutions) { n.add-b-solution(s); } }
59 ProofNode(e): add-f-sol-monitor(n)
60  { f-sol-monitors.add(n);  foreach (s in f-solutions) { n.add-f-solution(s); } }
```

Notes: On lines 44, 48, 51, 54, 58, 60, there are loops going over a set of nodes, solutions, or solution monitors. One should make a local snapshot copy and go over this local copy, since propagating solutions may result in new additions. ∎

To answer the three kinds of queries listed at the beginning of Section 3, we can use the following three procedures.

ProofGraph DBidirectional(e_0, D_0)

```
{ g = new ProofGraph();  nb = g.addNode(e0);  nb.b-activate();
   nf = g.addNode(D0);  bf.f-activate();  g.run();  return g; }
```

ProofGraph DBackward(e_0)

```
{ g = new ProofGraph();  n = g.addNode(e0);  n.b-activate();  g.run();  return g; }
```

ProofGraph DForward(D_0)

```
{ g = new ProofGraph();  n = g.addNode(D0);  n.f-activate();  g.run();  return g; }
```

The three procedures all use Algorithm 4 and differ only in their initial configurations. More generally, one can create a new proof graph, add some nodes, activate them, run the search algorithm, and return the proof graph as result. We call such a process one *run* of the distributed search algorithm. Each run R returns a proof graph as the result, which we denote PG_R. Each run asks for credentials while executing lines 14, 33, 36. We say that a run *discovers* a credential if it gets the credential by (executing) line 14 or line 33, and use $\mathcal{C}(R)$ to denote the set of credentials discovered by a run R. For the following reason, we do not consider credentials found by line 36 (L36 for short) to be discovered: Every credential found by L36 is a type-4 credential; L38 only retrieves the intersection in its body, and does not construct the credential edge corresponding to it. If such a credential actually contributes to solving the goal, it will be discovered on L33 and used on L34.

4.2 Correctness of the Distributed Chain Discovery Algorithm

We now prove that the distributed algorithm is correct in the following sense: the proof graph PG_R, generated by any run R of the algorithm, is sound and complete with respect to the credential graphs of $\mathcal{C}(R)$, that is, of the set of credentials the run R discovers.

Theorem 6 (Soundness of Distributed Search) *For any run R of the distributed search algorithm, the following two propositions hold.*

1. *If a node e_0 in PG_R has an entity D_0 as a backward solution, there exists a path $e_0 \overset{*}{\leftarrow} D_0$ in PG_R; furthermore, $e_0 \overset{*}{\leftarrow} D_0$ also exists in some credential graph of $\mathcal{C}(R)$.*
2. *If a node e_0 in PG_R has a role $A_0.r_0$ as a forward solution, then there exists a path $A_0.r_0 \overset{*}{\leftarrow} e_0$ in PG_R; furthermore, $A_0.r_0 \overset{*}{\leftarrow} e_0$ exists in some credential graph of $\mathcal{C}(R)$.*

Proof. *Observation 1: If a node e_0 in PG_R has an entity D_0 as a b-solution, then there is a path $e_0 \overset{*}{\leftarrow} D_0$ in PG_R.* In PG_R, a node e can get a backward solution in two ways. One, when e is an entity B, the node gets B as a backward solution when being processed (L22). This is the only case when a new backward solution is introduced to PG_R. Two, the node e gets b-solutions when it is a b-sol-monitor on another node e' and solutions are propagated from e' to e (L51 and L58). The node e is added as a b-sol-monitor on e' (L10) only when an edge $e \leftarrow e'$ is added to the proof graph. Thus, D_0 can be introduced as a solution for the first time only at the node D_0, and it can be propagated to e_0 only by traversing a path $e_0 \overset{*}{\leftarrow} D_0$ in PG_R.

Observation 2: If e_0 has $A_0.r_0$ as a f-solution, then there is a path $A_0.r_0 \overset{*}{\leftarrow} e_0$ in PG_R; in addition, if e_0 has a partial f-solution $\langle f_1 \cap \cdots \cap f_k, j\rangle$, then there is a path $f_j \overset{*}{\leftarrow} e_0$ in PG_R. This follows from arguments similar to those in observation 1.

Observation 3: Let Q be the set of role expressions in PG_R but not in $G_{\mathcal{C}(R)}$. If $e_2 \overset{*}{\leftarrow} e_1$ exists in PG_R, then it exists in $G_{\mathcal{C}(R):Q}$. We use induction on the order of edges being added to PG_R to show that every edge in PG_R also exists in $G_{\mathcal{C}(R):Q}$. An edge may be added to PG_R by (executing) line 15, 20, 28, 34, 40, or 56.

case 1: If an edge is added by L15 or L34, it has the form $A.r \leftarrow e$, and $A.r \longleftarrow e \in \mathcal{C}(R)$, so the edge is added to $G_{\mathcal{C}(R):Q}$ by closure property 1.

case 2: If an edge is added by L20, it has the form $A.r_1.r_2 \leftarrow B.r_2$, and $A.r_1$ has B as a b-solution. By observation 1, $A.r_1 \stackrel{*}{\leftarrow} B$ exists in PG_R. By induction hypothesis, $A.r_1 \stackrel{*}{\leftarrow} B$ exists in $G_{\mathcal{C}(R):Q}$, so $A.r_1.r_2 \leftarrow B.r_2$ is in $G_{\mathcal{C}(R):Q}$ by closure property 2.

case 3: If an edge is added by L40, it has the form $A.r_1.r_2 \leftarrow B.r_2$, and B has $A.r_1$ as a f-solution. By observation 2, $A.r_1 \stackrel{*}{\leftarrow} B$ exists in PG_R. This case now proceeds like case 2.

case 4: If an edge is added by L28, it has the form $f_1 \cap \cdots \cap f_k \leftarrow D$, and for each $j \in [1..k]$, f_j has D as a b-solution. By observation 1, for each $j \in [1..k]$, $f_j \leftarrow D$ exists in PG_R. So, by induction hypothesis, each $f_j \leftarrow D$ exists in $G_{\mathcal{C}(R):Q}$, and hence $f_1 \cap \cdots \cap f_k \leftarrow D$ also exists in $G_{\mathcal{C}(R):Q}$ by closure property 3.

case 5: If an edge is added by L56, it has the form $f_1 \cap \cdots \cap f_k \leftarrow D$, and D has k partial f-solutions of $f_1 \cap \cdots \cap f_k$. By observation 2, for each $j \in [1..k]$, $f_j \leftarrow D$ exists in PG_R, and this case proceeds like case 4.

The theorem now follows from the three observations. ∎

To assist us in defining and proving a notion of completeness, we introduce some terminology. We say that a node is *b-activated* (respectively, *f-activated*) by a run R, if the method b-activate() (f-activate()) was called on the node during the run. It is not difficult to see that a node is b-activated (respectively, f-activated) by a run R if and only if the node's b-proc-state (f-proc-state) value is "processed" in PG_R. So we also say that a node is b-activated (respectively, f-activated) in the proof graph PG_R.

The following completeness theorem roughly says that, any path in $G_{\mathcal{C}(R):Q}$ also exists in PG_R if the run R is set up to look for the path. In addition, if we have a path in PG_R, then the solutions propagate correctly, so that one can answer queries by looking up solutions. This theorem guarantees that if a run can discover all credentials in a chain, it can discover the chain.

Theorem 7 (Local Completeness of Distributed Search) *If there exists a path $e_2 \stackrel{*}{\leftarrow} e_1$ in $G_{\mathcal{C}(R):Q}$, then for any run R, the following two propositions hold:*

1. *If e_2 is b-activated in PG_R, then $e_2 \stackrel{*}{\leftarrow} e_1$ exists in PG_R, e_1 is b-activated, and all b-solutions on e_1 are also b-solutions on e_2.*
2. *If e_1 is f-activated in PG_R, then $e_2 \stackrel{*}{\leftarrow} e_1$ exists in PG_R, e_2 is f-activated, and all f-solutions on e_2 are also f-solutions on e_1.*

Proof. *Observation 1: If a path $e_2 \stackrel{*}{\leftarrow} e_1$ exists in PG_R, and e_2 is b-activated, then e_1 is also b-activated.* We show that given an edge $e'' \leftarrow e'$ in PG_R, if e'' is b-activated, then so is e'. The observation will then follow by induction on the length of the path. If e''.b-activate() is called before the addition of the edge $e'' \leftarrow e'$, then e'.b-activate() is called while adding the edge (L10). If e''.b-activate() is called after the addition of the edge, then $e'' \leftarrow e'$ exists when e''.b-activate() is called, and so e'.b-activate() is called on L44.

Observation 2: If a path $e_2 \stackrel{*}{\leftarrow} e_1$ exists in PG_R, and e_1 is f-activated, then e_2 is also f-activated. The arguments are similar to those in observation 1.

Observation 3: If a path $e_2 \stackrel{*}{\leftarrow} e_1$ exists in PG_R; and e_2 is b-activated, then all backward solutions of e_1 are also backward solutions of e_2. We show that when an edge $e'' \leftarrow e'$ exists in

PG_R, and e'' is b-activated, then all backward solutions on e' are also backward solutions of e''. The observation will then follow by induction on the length of the path. When $e'' \leftarrow e'$ is first added, e'' is added as a b-sol-monitor on e'. If a b-solution of e' is added before the addition of this b-sol-monitor, then the b-solution is propagated when the b-sol-monitor is added to e' (L58). If a b-solution to e' is added after that, then the b-solution is propagated when it is added (L51).

Observation 4: If a path $e_2 \overset{*}{\leftarrow} e_1$ exists in PG_R, and e_1 is f-activated, then all forward solutions of e_2 are also forward solutions of e_1. The arguments are similar to those in observation 3.

We now prove proposition 1 by showing that when a path $e_2 \overset{*}{\leftarrow} e_1$ exists in $G_{\mathcal{C}(R):Q}$, and e_2 is b-activated in PG_R, then $e_2 \overset{*}{\leftarrow} e_1$ also exists in PG_R. This, together with observations 1 and 3, proves proposition 1.

We use induction over the steps of the construction of $E^i_{\mathcal{C}(R):Q}$ given in Definition 2. In the base case, for each credential edge in $G_{\mathcal{C}(R):Q}$, there exists a credential in $\mathcal{C}(R)$. The credential is discovered by L14 or L33, and an edge is added to PG_R immediately by L15 or L34.

For the step, $e_2 \overset{*}{\leftarrow} e_1$ is in $E^{i+1}_{\mathcal{C}(R):Q}$. It suffices to show that the (unique) edge added to $E^i_{\mathcal{C}(R):Q}$ to construct $E^{i+1}_{\mathcal{C}(R):Q}$ also exists in PG_R, provided it is contained in $e_2 \overset{*}{\leftarrow} e_1$.

When the new edge is added by closure property 2, it has the form $A.r_1.r_2 \leftarrow B.r_2$, and $A.r_1 \overset{*}{\leftarrow} B$ exists in $E^i_{\mathcal{C}(R):Q}$. Then $e_2 \overset{*}{\leftarrow} e_1$ can be decomposed into $e_2 \overset{*}{\leftarrow} A.r_1.r_2 \leftarrow B.r_2 \overset{*}{\leftarrow} e_1$, with $e_2 \overset{*}{\leftarrow} A.r_1.r_2$ in $E^i_{\mathcal{C}(R):Q}$. If e_2 is b-activated, by induction hypothesis and observation 1, so is $A.r_1.r_2$; b-processing $A.r_1.r_2$ creates $A.r_1$ (L18). By the existence of $A.r_1 \overset{*}{\leftarrow} B$, the induction hypothesis, and observation 1, B is also b-activated. This means that B has B as a solution (L22). So by observation 3, $A.r_1$ has B as a b-solution. Because $A.r_1.r_2$ is activated, a BLinkingMonitor for $A.r_1.r_2$ is added on $A.r_1$ (L18). So when B is added as a solution on $A.r_1$, $A.r_1.r_2 \leftarrow B.r_2$ is added by L20.

When the new edge is added by closure property 3, it has the form $f_1 \cap \cdots \cap f_k \leftarrow D$, and for each $j \in [1..k]$, $f_j \overset{*}{\leftarrow} D$ exists in $E^i_{\mathcal{C}(R):Q}$. In this case, $e_1 = D$, and $e_2 \overset{*}{\leftarrow} e_1$ can be decomposed into $e_2 \overset{*}{\leftarrow} f_1 \cap \cdots \cap f_k \leftarrow D$. If e_2 is b-activated, by induction hypothesis and observation 1, so is $f_1 \cap \cdots \cap f_k$; processing $f_1 \cap \cdots \cap f_k$ creates f_j and b-activates them. By induction hypothesis and observations 1 and 3, D is b-activated and is added as a solution to each f_j, and so the linking monitor for $f_1 \cap \cdots \cap f_k$ receives D k times, and $f_1 \cap \cdots \cap f_k \leftarrow D$ is added by L28.

Similarly, using observations 2 and 4, we can prove proposition 2. ∎

As a corollary of this theorem, if there is a path $e \overset{*}{\leftarrow} D$ in $G_{\mathcal{C}(R):Q}$, and e is b-activated in PG_R, then e has D as a backward solution. Similarly, if there is a path $A.r \overset{*}{\leftarrow} D$ in $G_{\mathcal{C}(R)}$, and D is f-activated, then D has $A.r$ as a forward solution.

4.3 Traversability of Distributed Chains and Their Discovery

When credential storage is distributed, the distributed search algorithm may not be able to find a credential chain because of credential storage. We introduce notions of traversability to formalize the three different ways in which distributed chains can be located and assembled, depending on the storage characteristics of their constituent credentials. We call the three notions, *forward traversability*, *backward traversability*, and *confluence*, respectively.

Let us first illustrate these three notions through a chain $\langle A.r \twoheadleftarrow D\rangle$ that consists only of credential edges. In this case, $\langle A.r \twoheadleftarrow D\rangle = A.r \overset{*}{\leftarrow} D$. Suppose that all credentials in $A.r \overset{*}{\leftarrow} D$ are stored with their subjects. We call $A.r \overset{*}{\leftarrow} D$ *forward traversable* because $\mathsf{DForward}(D)$ can discover the path, as follows. Obtain from D the first credential of the path and, with it, the identity (and hence the location) of the issuer of that credential. That issuer is the subject of the next credential. By visiting each successive entity in the path and requesting credentials from them, each credential in the path can be obtained. Similarly, if all credentials in $A.r \overset{*}{\leftarrow} D$ are stored with their issuers, the path is *backward traversable*, and can be discovered by $\mathsf{DBackward}(A.r)$. Credentials in the path can be collected from entities starting with A and working from issuers to subjects. When some credentials are stored with issuers and others with subjects, one can do a bidirectional search, working from both ends towards the middle. To be able to assemble the path $A.r \overset{*}{\leftarrow} D$ using $\mathsf{DBidirectional}(A.r, D)$, it suffices that $A.r \overset{*}{\leftarrow} D$ can be decomposed into two subpaths, the one starting from D forward traversable and the one ending at $A.r$ backward traversable; in this case, we say that the path $A.r \overset{*}{\leftarrow} D$ is *confluent*.

The following definition extends these intuitions to chains that contain derived edges as well as credential edges.

Definition 4 (Traversability and Confluence)

Let $\mathcal{C}$ be a set of credentials, and $\langle e_2 \twoheadleftarrow e_1 \rangle$ a chain such that $\mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle \subseteq \mathcal{C}$.

The chain $\langle e_2 \twoheadleftarrow e_1 \rangle$ has the same traversability as the path $e_2 \overset{*}{\leftarrow} e_1$.

A path $e_2 \overset{*}{\leftarrow} e_1$ is:

Forward traversable if it is empty ($e_1 = e_2$), or it consists entirely of forward traversable edges;

Backward traversable if it is empty, or it consists entirely of backward traversable edges;

Confluent if it is empty, or it can be decomposed into $e_2 \overset{*}{\leftarrow} e'' \leftarrow e' \overset{*}{\leftarrow} e_1$ where $e' \overset{*}{\leftarrow} e_1$ is forward traversable, $e_2 \overset{*}{\leftarrow} e''$ is backward traversable, and $e'' \leftarrow e'$ is confluent.

A credential edge is:

Forward traversable if the credential it represents is held by each subject of the credential;

Backward traversable if the credential it represents is held by the issuer of the credential;

Confluent if it is forward or backward traversable.

A derived link edge, $A.r_1.r_2 \leftarrow B.r_2$ is:

Forward traversable if its supporting path $A.r_1 \overset{*}{\leftarrow} B$ in $\langle e_2 \twoheadleftarrow e_1 \rangle$, is forward traversable;

Backward traversable if $A.r_1 \overset{*}{\leftarrow} B$ is backward traversable;

Confluent if $A.r_1 \overset{*}{\leftarrow} B$ is confluent;

A derived intersection edge, $f_1 \cap \cdots \cap f_k \leftarrow D$ is:

Forward traversable if (a) there exists an $\ell \in [1..k]$ with $f_\ell \overset{*}{\leftarrow} D$ forward traversable, and (b) for each $j \in [1..k]$, $f_j \overset{*}{\leftarrow} D$ is confluent;

Backward traversable if (a) there exists an $\ell \in [1..k]$ with $f_j \overset{*}{\leftarrow} D$ backward traversable, and (b) for each $j \in [1..k]$, $f_j \overset{*}{\leftarrow} D$ is confluent;

Confluent if for each $j \in [1..k]$, $f_j \overset{*}{\leftarrow} D$ is confluent;

Note that if an edge or a chain is forward or backward traversable, it is also confluent.

We are now ready to state and prove the second part of the correctness argument of the distributed search algorithm.

Theorem 8 (Completeness of Distributed Search) *Given a set $\mathcal{C}$ of credentials that are stored in a distributed manner, and a chain $\langle e_2 \twoheadleftarrow e_1 \rangle$ in a credential graph of $\mathcal{C}$, the following three propositions hold.*

1. *If $\langle e_2 \twoheadleftarrow e_1 \rangle$ is backward traversable, then for any run R of the distributed search algorithm: if e_2 is b-activated in PG_R, then $\mathcal{C}(R) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$. In particular, since e_2 is b-activated by the run* $\mathsf{DBackward}(e_2)$*, $\mathcal{C}(\mathsf{DBackward}(e_2)) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$.*
2. *If $\langle e_2 \twoheadleftarrow e_1 \rangle$ is forward traversable, then for any run R: if e_1 is f-activated in PG_R, then $\mathcal{C}(R) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$. In particular, $\mathcal{C}(\mathsf{DForward}(e_1)) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$.*
3. *If $\langle e_2 \twoheadleftarrow e_1 \rangle$ is confluent, then for any two runs R_1 and R_2: if e_1 is f-activated in R_1 and e_2 is b-activated in R_2, then $\mathcal{C}(R_2) \cup \mathcal{C}(R_1) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$. In particular, $\mathcal{C}(\mathsf{DBackward}(e_2)) \cup \mathcal{C}(\mathsf{DForward}(e_1)) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$.*[7]

See Appendix A.5 for the proof. Theorems 8 and 7 together ensure that any traversable chain will be found by the distributed search algorithm.

5 A Type System for Distributed Credential Storage

If we assume that all credentials are stored with their issuers, then all chains are backward traversable. Similarly, if we assume that all credentials are stored with their subjects, all chains are forward traversable. As we argued in Section 1.1, neither assumption is realistic in practice. Yet some constraints must be imposed on credential storage, or else some chains cannot be discovered without trying to contact every entity in the system. For example, consider a chain $A_1.r \leftarrow A_2.r \leftarrow A_3.r$. If both credentials are stored only with A_2, then knowing only $A_1.r$ and $A_3.r$, one cannot discover the chain. One observation from this simple example is that if we require that all credentials defining roles using the role name r have the same storage characteristics, the chain is either backward or forward traversable. Capitalizing on this observation, we introduce in this section a type system for distributed credential storage, the important feature of which is that, given a set of well typed credentials, every chain $\langle e \twoheadleftarrow e_1 \rangle$ is confluent if e is well typed. The type system provides this global guarantee by imposing local constraints on each credential. We then go on to explore various ways these constraints might be relaxed, and analyze their advantages and disadvantages. We also present a practical mechanism for achieving agreement about type assignments.

5.1 A Type System for Credential Storage

In our storage type system, each role name r has two types: an issuer-side type and a subject-side type. On the issuer side, each role name has one of three type values: *issuer-traces-none*, *issuer-traces-def*, and *issuer-traces-all*. If a role name r is issuer-traces-def, then from any entity A one can find all credentials defining $A.r$. In other words, A must store all credentials defining $A.r$. However, this does not guarantee that one can find all members of $A.r$. For instance, we might have $A.r \longleftarrow B.r_1$, with r_1 issuer-traces-none. This motivates the stronger type: issuer-traces-all. A role name r being issuer-traces-all implies not only that A stores all credentials defining $A.r$,

[7] This proposition implies that $\mathcal{C}(\mathsf{DBidirectional}(e_2, e_1)) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$. We prove the stronger result because it allows two parties to do search separately without knowing each other, and then combine credentials and discover chains. This is useful for automated trust negotiation, *e.g.*, [20].

but also that any credential $A.r \longleftarrow B.r_1$ has $B.r_1$ issuer-traces-all, as well. By generalizing this basic idea, we obtain the result that, for any entity A, using (distributed) backward search, one can determine all the members of $A.r$. (In this section, all searches are distributed, and so we omit the word distributed from now on.)

On the subject side, each role has one of two type values: *subject-traces-none* and *subject-traces-all*. If a role name r is subject-traces-all, the intention is that from any entity D, forward search can find all roles of the form $A.r$ such that D is a member of $A.r$. For instance, if there is a credential $A.r \longleftarrow B.r_1$, then B has to store this credential. Moreover, the type rules for credentials will require that r_1 is also subject-traces-all, if this credential is to be well typed. These intuitions are captured by the definition of well typed credentials in Definition 7 below.

There are three values for the issuer-side type and two values for the subject-side type, yielding six combinations. Among them, one combination is illegal; if a role name r is both issuer-traces-none and subject-traces-none, no one is required to store credentials defining $A.r$, and so such credentials cannot be discovered. This leads to the notion of well typed role names.

Definition 5 (Well-typedness of Role Names)

- A role name is *strongly well typed* if it is issuer-traces-all or subject-traces-all.
- A role name is *weakly well typed* if it is both issuer-traces-def and subject-traces-none.
- A role name is *well typed* if it is strongly well typed or weakly well typed.
- A role name is *ill-typed* if it is not well typed. In other words, a role name is ill-typed if it is both issuer-traces-none and subject-traces-none.

We now extend this notion of well-typedness to role expressions and then define the notion of well typed credentials. As we show below in Section 5.3, Definitions 5, 6, and 7 together guarantee that when credentials are well typed, the following three propositions hold:

1. If a role expression e is well typed, then any chain $\langle e \twoheadleftarrow e_1 \rangle$ is confluent, and so from any member D of e, one can discover $\langle e \twoheadleftarrow D \rangle$ by doing bidirectional search.
2. If e is issuer-traces-all, then every chain $\langle e \twoheadleftarrow e_1 \rangle$ is backward traversable, and so one can find all members of e by doing backward search from e.
3. If e is subject-traces-all, then every chain $\langle e \twoheadleftarrow e_1 \rangle$ is forward traversable, and so from any member D of e, one can discover $\langle e \twoheadleftarrow D \rangle$ by doing forward search from D.

Definition 6 (Well-typedness of role expressions)

- An entity A is both issuer-traces-all and subject-traces-all.
- A role $A.r$ has the same type as r.
- A linked role $A.r_1.r_2$ is

$$\begin{cases} \text{issuer-traces-all} & \text{if both } r_1 \text{ and } r_2 \text{ are issuer-traces-all} \\ \text{subject-traces-all} & \text{if both } r_1 \text{ and } r_2 \text{ are subject-traces-all} \\ \text{weakly well typed} & \text{otherwise, if either } r_1 \text{ is issuer-traces-all and } r_2 \text{ is well typed,} \\ & \text{or } r_1 \text{ is well typed and } r_2 \text{ is subject-traces-all} \\ \text{ill-typed} & \text{otherwise} \end{cases}$$

- An intersection $f_1 \cap \cdots \cap f_k$ is

$$\begin{cases} \text{issuer-traces-all} & \text{if there exists an } f_\ell \text{ that is issuer-traces-all, and all } f_j\text{'s are} \\ & \text{well typed} \\ \text{subject-traces-all} & \text{if there exists an } f_\ell \text{ that is subject-traces-all, and all } f_j\text{'s are} \\ & \text{well typed} \\ \text{weakly well typed} & \text{if all } f_j\text{'s are weakly well typed} \\ \text{ill-typed} & \text{otherwise} \end{cases}$$

The intuition behind the rule for a well-typed linked role $A.r_1.r_2$ is as follows. To discover a chain $\langle A.r_1.r_2 \twoheadleftarrow D\rangle$, one needs to discover two subchains, $\langle A.r_1 \twoheadleftarrow B\rangle$ and $\langle B.r_2 \twoheadleftarrow D\rangle$, for some entity B. If both r_1 and r_2 are issuer-traces-all, then both subchains will be shown to be backward traversable, so one can discover $\langle A.r_1 \twoheadleftarrow B\rangle$ by starting from $A.r_1$, searching to B, then continuing from $B.r_2$ to D. If both r_1 and r_2 are subject-traces-all, both subchains are forward traversable, starting from D, and proceeding through $B.r_2$ and B to $A.r_1$. Now, if $\langle A.r_1 \twoheadleftarrow B\rangle$ is backward traversable, $\langle B.r_2 \twoheadleftarrow D\rangle$ can be discovered by bidirectional search from both $B.r_2$ and D, and so only needs to be confluent. Similarly, if $\langle B.r_2 \twoheadleftarrow D\rangle$ is forward traversable, $\langle A.r_1 \twoheadleftarrow B\rangle$ only needs to be confluent.

Definition 7 (Well Typed Credentials)

A credential $A.r \longleftarrow e$ is *structurally well typed* if the following three conditions are satisfied:

1. Both $A.r$ and e are well typed.
2. If $A.r$ is issuer-traces-all, e must also be issuer-traces-all.
3. If $A.r$ is subject-traces-all, e must also be subject-traces-all.

A credential $A.r \longleftarrow e$ is *well typed* if it is structurally well typed and satisfies the following two *storage requirements*:

- If $A.r$ is issuer-traces-def or issuer-traces-all, A stores this credential.
- If $A.r$ is subject-traces-all, every subject of this credential stores this credential.

5.2 Examples of Storage Types

We now discuss some examples of storage types.

Example 5 Let us consider a type assignment for credentials in Example 3. The corresponding credential chain $\langle \text{EPub.spdiscount} \twoheadleftarrow \text{Alice}\rangle$ is shown in Figure 4.
One appropriate type assignment is as follows:

$$\begin{cases} \text{spdiscount, preferred, university} & \text{are} & \text{issuer-traces-def and subject-traces-none} \\ \text{accredited, student, member} & \text{are} & \text{issuer-traces-none and subject-traces-all} \end{cases}$$

Under this type assignment, all the credentials are structurally well typed. To satisfy the storage requirements for well typed credentials, credential (1) is stored with EPub, (2) and (3) are stored with EOrg, (4) is stored with StateU, (5) is stored with RegistrarB, and both (6) and (7) are stored with Alice. One can verify that the chain $\langle \text{EPub.discount} \twoheadleftarrow \text{Alice}\rangle$ in Figure 4 can be discovered by doing bidirectional search.

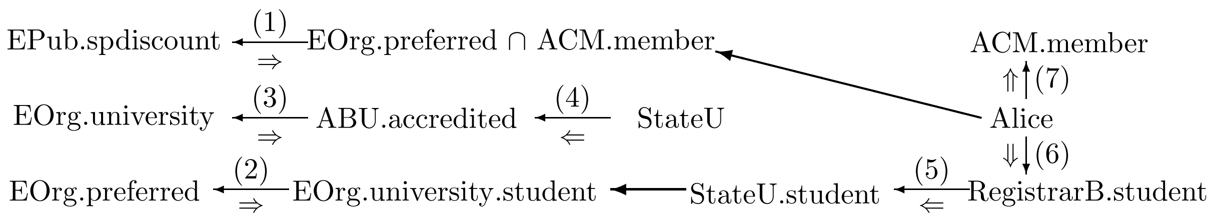


Figure 4: The chain ⟨EOrg.discount ↞ Alice⟩ discovered by DBidirectional(EOrg.discount, Alice), with the set of credentials in Example 3, typed as in Example 5. Edges with numbers are credential edges; the numbers above (or to the right of) edges correspond to the credential numbers in Example 3. Double arrows originate at a role expression whose base stores the corresponding credential. Thus, if the double arrow has the same direction as the edge, the credential can be discovered by forward search. Otherwise, the credential can be discovered by backward search.

Now consider Example 4, in which we also have millions of credentials about other universities and students. Under this type assignment, DBidirectional(EOrg.discount, Alice) does not touch those credentials. This is because backward search stops at ACM.member, EOrg.university.student, and ABU.accredited, since credentials defining ABU.accredited and ACM.member are not stored with issuers. This shows that distributed chain discovery is practical with appropriate storage type assignment. ∎

Continuing the discussions in Example 5, we now show some type assignments that make the chain ⟨EPub.spdiscount ↞ Alice⟩ unable to be discovered. We also show that these assignments make one or more credentials not structurally well typed, and so are "prevented" by the type system.

One assignment is obtained from that in Example 5 by making university issuer-traces-none and subject-traces-all, and making accredited issuer-traces-def and subject-traces-none. This assignment causes credentials (3) and (4) to be stored only with ABU, making it impossible to discover the subchain ⟨EOrg.university ↞ StateU⟩. However, it would also make credential (3) not structurally well typed, since EOrg.university is subject-traces-all, but the body of the credential, ABU.accredited, is not.

Another assignment is obtained from Example 5 by making student issuer-traces-all (or issuer-traces-def) and subject-traces-none. This results in both (4) and (5) being stored with StateU, making it impossible to discover the subchain ⟨EOrg.university.student ↞ RegistrarB.student⟩. However, according to Definition 6, EOrg.university.student is ill-typed, since university is weakly well typed and student is not subject-traces-all; as a result, credential (2) is not structurally well typed.

Now we consider two assignments that are correct, but less practical. The first is again a variant on Example 5. It changes the type of university and of accredited to be issuer-traces-all, and it makes student issuer-traces-def. This results in credentials (3), (4), and (5) being stored with their issuers. Although correct, this assignment is not as practical as the one in Example 5, because it requires backward processing of ABU.accredited and EOrg.university.student. In the scenario in Example 4, credentials about other universities and students are also discovered, resulting in a very large proof graph.

Another impractical assignment makes all role names in Example 5 subject-traces-all. In this case, all credentials must be stored with their subjects. For instance, credential (3) would be stored by ABU. In the example, credential (3) is in effect a local policy of EOrg, and of no concern to ABU. Neither party may consider such an arrangement reasonable.

This discussion illustrates that the design of type assignments is subject to pitfalls of at least three kinds. First, assignments inherently preclude certain credentials being well-typed. (This is the fundamental mechanism through which the type system guarantees traversability.) Thus type assignments must be designed to avoid precluding credentials that are needed. Second, well designed type assignments can significantly improve search performance. Third, the storage requirements must be agreeable to the parties that must implement them. Our conclusion is that type-assignment design should be left to experts. We return to this issue in Sections 5.5 and 5.6 below.

5.3 Traversability of Well Typed Chains

In this section we show that, given a set of well typed credentials whose storage is distributed, one can answer the three forms of queries enumerated at the beginning of Section 3 by using the distributed search algorithm in Section 4.

Theorem 9 (Traversability of Well Typed Chains) *Given a set $\mathcal{C}$ of well typed credentials, and any chain $\langle e \twoheadleftarrow e_1 \rangle$ in a credential graph $G_{\mathcal{C}:Q}$ of $\mathcal{C}$, the following propositions hold:*

1. *If e is well typed, $\langle e \twoheadleftarrow e_1 \rangle$ is confluent.*
2. *If e is issuer-traces-all, $\langle e \twoheadleftarrow e_1 \rangle$ is backward traversable.*
3. *If e is subject-traces-all, $\langle e \twoheadleftarrow e_1 \rangle$ is forward traversable.*

See Appendix A.6 for the proof. From Theorems 1, 2, 6, 7, 8, and 9, we have the following theorem.

Theorem 10 *Given a set $\mathcal{C}$ of well typed credentials, the following three propositions hold:*

1. *Given an entity D and a role expression e that is well typed, one can use $\mathsf{DBidirectional}(D, e)$ to determine whether $D \in \mathsf{expr}[\mathcal{S}_\mathcal{C}](e)$.*
2. *Given a role expression e that is issuer-traces-all, one can use $\mathsf{DBackward}(e)$ to determine $\mathsf{expr}[\mathcal{S}_\mathcal{C}](e)$.*
3. *Given an entity D, one can use $\mathsf{DForward}(D)$ to determine all the roles $A.r$ such that $A.r$ is subject-traces-all and $D \in \mathcal{S}_\mathcal{C}(A.r)$.*

Proof. We prove only proposition 1. The proof of propositions 2 and 3 are similar.

Observation 1: $D \in \mathsf{expr}[\mathcal{S}_\mathcal{C}](e)$ if and only if there exists a path $e \overset{*}{\leftarrow} D$ in $G_{\mathcal{C}:\{e\}}$. This follows from the soundness and completeness of credential graphs (Theorems 1 and 2).

Let $R = \mathsf{DBidirectional}(D, e)$, we have the following two observations.

Observation 2: A path $e \overset{*}{\leftarrow} D$ exists in $G_{\mathcal{C}:\{e\}}$ if and only if it also exists in $G_{\mathcal{C}(R):\{e\}}$. If a path from D to e does not exist in $G_{\mathcal{C}:\{e\}}$, then clearly it does not exist in $G_{\mathcal{C}(R):\{e\}}$, since $\mathcal{C}(R) \subseteq \mathcal{C}$. If $e \overset{*}{\leftarrow} D$ exists in $G_{\mathcal{C}:\{e\}}$, then by Theorem 9, the chain $\langle e \twoheadleftarrow D \rangle$ must be confluent. From the completeness of distributed search (Theorem 8), $\mathcal{C}(R) \supseteq \mathcal{C}\langle e \twoheadleftarrow D \rangle$. Therefore $e \overset{*}{\leftarrow} D$ exists in $G_{\mathcal{C}(R):\{e\}}$.

Observation 3: A path $e \xleftarrow{*} D$ exists in $G_{\mathcal{C}(R):\{e\}}$ if and only if D is a backward solution on e in PG_R. This follows from the soundness and completeness of distributed search (Theorems 6 and 7).

Proposition 1 follows from the above three observations. ∎

5.4 Discussions and Extensions to the Storage Type System

One obvious thing to note about the storage type system is that there are three issuer-side type values and only two subject-side type values. One may ask why is there not a value subject-traces-def. We now explain the reason. The goal of the type system is to ensure that one can discover a chain once both ends are known. Consider a path $A.r \leftarrow B.r_1 \leftarrow D$; to discover it, starting from $A.r$ and D, a search has to be able to move one step, either backward from $A.r$ or forward from D, to find $B.r_1$. To enable a search to move one step backward from $A.r$, credentials defining $A.r$ need to be stored with A. To enable moving one step forward from D, the credential $B.r_1 \longleftarrow D$ needs to be subject-stored. Suppose we did add subject-traces-def, and assigned it as the type of r. Neither of the above one-step moves would then be guaranteed possible. By contrast, the stronger subject-traces-all type imposes structural requirements on well-typed credentials, as well as storage requirements. Specifically, if r is subject-traces-all, r_1 must be too, by well-typedness of $A.r \longleftarrow B.r_1$, thus ensuring that $B.r_1 \longleftarrow D$ is stored by D.

One also may ask, if two subject-side type values suffice, why have three issuer-side values? We need the stronger issuer-traces-all value in addition to issuer-traces-def because of linked roles. For $A.r_1.r_2$ to be well typed, it is insufficient to have both r_1 and r_2 issuer-traces-def; however r_2 can be issuer-traces-def when r_1 is issuer-traces-all.

In the rest of this section, we discuss two extensions to the storage type system.

5.4.1 Adding preference in tracing direction

As we have seen in Section 5.2, storage types can be used to limit search space as well as to guarantee discovering credentials. For example, if the role names, accredited and student, are subject-traceable (*i.e.*, subject-traces-all), rather than issuer-traceable (issuer-traces-all or issuer-traces-def), then searching is more efficient. In practice, a role name may be both issuer-traceable and subject-traceable; however, searching from one direction is often preferred. This observation motivates us to enhance the type system to encode this preference. When a role name is both issuer-traceable and subject-traceable, one can specify which direction is preferred for traversing credentials that define the role name. For example, ACM may maintain all membership credentials (*e.g.*, to enable finding all members when needed) and, at the same time, expect its members to store and to provide those credentials themselves when needed. In this case, we can specify the role name, member, to be both subject-traces-all and issuer-traces-all, with subject-tracing preferred. This way, backward searches that involve individual members can stop when they encounter ACM.member, thereby avoiding large fan-out.

5.4.2 Storing type-1 and other types of credentials differently

In Section 1.1, we briefly discussed an approach requiring that all type-1 credentials be stored with subjects and all other types of credentials be stored with issuers. We showed that this approach is not by itself sufficient to guarantee that credential chains can be discovered, due to the presence of linked roles. Nevertheless, this idea has some appeal. When a role can be defined by type-1 credentials as well as by other kinds of credentials, it may be desirable to allow all type-1 credentials to be stored with subjects and other credentials with issuers. This additional flexibility may be needed for some applications.

We introduce an extended type system to allow this flexibility, while still ensuring that chains can be discovered in the presence of linked roles. In this system, a role name can be *issuer-traces-none*, *issuer-traces-rule*, *issuer-traces-def*, or *issuer-traces-all* on the issuer side and *subject-traces-none*, *subject-traces-fact*, or *subject-traces-all* on the subject side. Both of these lists are in order of increasing strength of the associated storage requirements.

We need to change two things in Definitions 5, 6, and 7. First, a role name is weakly well typed when it is issuer-traces-def and subject-traces-none, or when it is issuer-traces-def and subject-traces-fact, or when it is issuer-traces-rule and subject-traces-fact. Second, the storage requirements for a credential $A.r \longleftarrow e$ to be well typed is changed as follows.

- The issuer A has to store the credential if r is issuer-traces-all or issuer-traces-def, or if r is issuer-traces-rule and e is not an entity.
- The subjects of the credential have to store it if r is subject-traces-all, or if r is subject-traces-fact and e is an entity.

Given this extended type system, Theorem 9 still holds. The proof is almost the same as that of Theorem 9. See Appendix A.7 for details.

5.5 Agreeing on Types and Meanings of Role Names

We now discuss some questions related to the practical use of the storage type system. Our type system raises the following question: How can entities agree on the type of a role name? A similar problem exists even without storage types. When an entity A defines $A.r$ to contain $B.r_1$, it needs to understand what B means by the role name r_1. Consider again the credentials in Example 1, whose credential graph we repeat here:

$$\text{EPub.discount} \xleftarrow{(1)} \text{EOrg.preferred} \xleftarrow{(2)} \text{StateU.student} \xleftarrow{(3)} \text{RegistrarB.student} \xleftarrow{(4)} \text{Alice.}$$

Given the role StateU.student, how does EOrg know what StateU means by student? Is it issued to students registered in any class, or only to students enrolled in a degree program? If EOrg does not know, how can EOrg issue a credential EOrg.preferred $\longleftarrow$ StateU.student? This is the problem of establishing a common vocabulary. Different entities need a common vocabulary before they can use each others' roles. This problem arises in all trust-management systems. However, name agreement is particularly critical in systems, like RT_0, that support attribute-based delegation. For instance, the expression EOrg.university.student only makes sense when universities use student for the same purpose.

We present the following scheme to achieve name agreement. The scheme is inspired by XML namespaces [5]. We introduce *application domain specification documents (ADSDs)*. Each ADSD defines a vocabulary. In the RT framework [17], ADSDs are used to define data types and parameter types of parameterized roles. Since role names in RT_0 cannot take parameters, an RT_0 vocabulary is just a suite of related role names. We can use ADSDs to also declare storage types of the role names. An ADSD generally should give natural-language explanations of these role names, including the conditions under which credentials defining them should be issued. A logical role name consists of two parts, a vocabulary id that uniquely identifies an ADSD, *e.g.*, a URI to the ADSD, and a role id that is declared in the ADSD.

Example 6 Consider the credentials in Example 3. There we just use role ids for role names and ignore the issues of which ADSDs they belong to. These role names are likely to be declared in different ADSDs. One possible scenario is that there are four ADSDs involved.

1. An ADSD local to EPub, which declares spdiscount.
2. An ADSD about EOrg's policies, created by EOrg; it declares preferred and university.
3. An ADSD about universities and students, which may be created by ABU or others; it declares accredited and student.
4. An ADSD about ACM members, which is created by ACM, it declares member.

Other scenarios are also possible. In the credential ACM.member $\longleftarrow$ Alice, the role name, member, may refer to an ADSD created jointly by several professional organizations. In the credential StateU.student $\longleftarrow$ RegistrarB.student, the student in the body may refer to an ADSD created locally by StateU, while the one in the head refers to an ADSD created by ABU. In that case, the two occurrences of student refer to two different logical role names, and they may have different storage types.

When there are multiple ADSDs about university students, a university can freely choose which ADSD to use when it issues credentials. A university can issue multiple student credentials using different ADSDs, or, if appropriate, it can issue vocabulary-mapping credentials, each one translating the student role id in one ADSD to the student role id in another ADSD. ∎

Each ADSD defines a vocabulary. The notion of vocabularies is complimentary to the notion of localized name spaces for roles. Each addresses a distinct role name-space issue. A role consists of an issuer, a vocabulary id, and a role id. Each issuer of credentials has its own localized name space within which the issuer has sole authority to define role members. Each vocabulary id specifies a distinct vocabulary of role names, thus enabling different issuers to agree on a common understanding of those role names.

5.6 Whether to Allow Localized Storage Types

Our type system requires consistent storage of credentials across roles that use the same role name. In principle, this requirement can be relaxed, which might seem advantageous to support greater decentralization. In this approach, $A.r$ and $B.r$ can have different types. It remains true that when each credential is well typed, credential chains can be discovered. However, in this section we argue that such an approach is inconsistent with the goals of having attribute-based delegation and distributed storage, and that it has other practical difficulties.

First and foremost, if different universities may assign different types to roles using the same role name, student, then the type-3 credential EOrg.preferred $\longleftarrow$ EOrg.university.student can be used only when EOrg.university is issuer-traces-all. This follows because now it is safe to assume only that B.student is weakly well typed. In our view, requiring the first part of every linked role (like EOrg.university) to be traversed backward is unacceptably restrictive. Moreover, because the type of the second part (student) is undetermined, the linked role, too, can at best be assumed to be weakly well typed, so EOrg.preferred cannot be issuer-traces-all. Consequently, one cannot use EOrg.preferred to start a linked role. This makes it impossible to implement long linked local names in SDSI.

Second, each entity A that issues credentials of the form $A.r \longleftarrow B.r_1$ has to have a consistent view of the storage type that B assigns to $B.r_1$. This requires much more infrastructure support than does supporting consistency at the role-name level. The latter can be supported through ADSDs in some sense for free, since ADSDs are needed by the RT framework for other reasons [17].

Third, type assignments guarantee traversability by imposing storage requirements and precluding certain credentials being well-typed. Also, some type assignments lead to more efficient searches than do others. For these reasons, we argue that type assignments should be designed by experts who are familiar with the application domain and anticipate certain credential templates. This is at odds with types being assigned at the level of individual roles.

6 Future and Related Work

In this section, we illustrate briefly the next step in our Role-based Trust-management framework work, and then discuss other future directions and related work.

6.1 Ongoing Related Work

As mentioned in Section 1, RT_0 is the first step in a series of Role-based Trust-management languages. In [17], four more components of RT are defined; they are RT_1, RT_2, RT^T, and RT^D. RT_1 extends RT_0 in allowing role names to take parameters and credentials to use variables. For example, the credential OS.fileop(delete, ?file) $\longleftarrow$ OS.owner(?file) can be used to express the policy that the operating system will let a file's owner delete the file. RT_2 adds to RT_1 logical objects, which allows one to group logically related resources together and to assign permissions about them together. RT^T provides manifold roles and role-product operators, which can express threshold and separation-of-duty policies. RT^D provides delegation of role activations, which can express selective use of capacities (role memberships) and delegation of these capacities. See [17] for more details.

Like most prior trust-management work, we assume in this paper that credentials are freely available to the agent responsible for making access control decisions. In general, credentials may contain sensitive information. To protect sensitive credentials while allowing them to be used in a decentralized environment, trust management-style access control can be applied to credentials, as to any other resource. Trust can be established between two entities in such a context through an iterative process of revealing credentials to one another, called a trust negotiation [21]. Concurrently with the design of RT, we are developing a system for automated trust negotiation that supports RT. A design supporting RT_0 is presented in [20], where additional references to work in this area can also be found. By taking advantage of the distributed chain discovery algorithm and the storage type system presented here, the trust negotiation system ensures that the two entities participating in the trust negotiation can discover and collect, either before or during the negotiation, all credentials required to construct each chain that is needed during trust negotiation.

6.2 Storage Types and Complete Information

Inferencing based on credentials stored in a distributed manner is often limited by not knowing whether all relevant credentials are present. Because trust management aims to deal with authorization in distributed systems, most TM languages are monotonic. In contrast, most work on access control models in centralized environments allow non-monotonicity in the form of negative authorizations or constraints.

Limiting the system to be monotonic ensures that, even without access to all credentials, if the credentials that are present indicate D is a member of $A.r$, it is certainly true. Missing credentials could make you unable to prove D is a member of $A.r$, but cannot lead you to conclude D is a member of $A.r$ erroneously. However, some policies are non-monotonic in nature. Examples include constraints in RBAC, such as mutually exclusive roles (*i.e.*, roles that cannot have common members), and cardinality of roles (*e.g.*, a role can have at most a certain number of members).

The storage type approach presented in this paper could be used to help solve this problem. The type system ensures we know who to contact to request the relevant credentials. So assuming they respond and we trust that they give us the credentials we ask for, we can assume that we obtain all the credentials that are relevant. In this context, it may be safe to use non-monotonic inference rules. It will be necessary to manage the trust issue. For instance, we may trust that some issuers will give us all relevant credentials, while not trusting some subjects to do the same. We do not claim that the storage type system solves the problem of having non-monotonic policies; nevertheless, we believe that it could be one useful tool.

6.3 Decentralized Construction of Proof Graphs

In Section 4, proof graph construction is centralized in one entity, although credentials can be stored in a distributed manner. An interesting extension to this is to have each entity maintain a part of the whole graph. In fact, such graphs can be used as a storage format for credentials.

In the following, we prove that the discovery problem of RT_0 is log-space P-complete; thus is "inherently sequential" [10]. As usual, P is the class of problems solvable in deterministic polynomial time. It is widely conjectured that problems that are P-complete are not in NC, which is the class of problems that can be solved on a Parallel Random Access Machine using $\log n^{O(1)}$ parallel time and $n^{O(1)}$ processors.

One problem that is log-space P-complete is the monotone circuit value problem (MCVP). A *monotonic circuit* β is a sequence $(\beta_0, \ldots, \beta_n)$, where each β_i is either an input, an AND gate $AND(j,k)$, or an OR gate $OR(j,k)$; and the 0,1 values of the inputs are given explicitly. $MCVP = \{\beta \mid \beta$ is a monotone circuit with the output value of $\beta_n = 0\}$. From [10], we have: $MCVP$ is log-space complete for P.

Theorem 11 *The discovery problem for RT_0 is log-space P-complete.*

Proof. We prove that RT_0, even without intersections, is log-space P-complete. This means that name reduction in SDSI 2.0 is also log-space P-complete. Given an instance of $MCVP$, β, construct a set of credentials $\mathcal{C}$ as follows. $\mathcal{C}$ has one entity: 1. Each β_i is mapped to a role $1.\beta_i$. If an input β_i has value 1, a credential $1.\beta_i \longleftarrow 1$ is added to $\mathcal{C}$; if the input has value 0, no credential is added. For each AND gate $\beta_i = AND(\beta_j, \beta_k)$, a credential $1.\beta_i = 1.\beta_j.\beta_k$ is added. For each OR gate $\beta_i = OR(\beta_j, \beta_k)$, two credentials $1.\beta_i \longleftarrow 1.\beta_j$ and $1.\beta_i \longleftarrow 1.\beta_k$ are added. Clearly, $\beta_n = 1$ if and only if the role $1.\beta_n$ has 1 as its member. The reduction only takes constant space. ∎

This result suggests that it is unlikely to obtain exponential speedup by applying polynomial processors to chain discovery. It remains worthwhile to investigate how much speedup can be obtained by performing distributed construction of proof graphs. It would be particularly interesting to understand the tradeoff between communication and computation costs. For this purpose, PRAM is not an appropriate computation model, since it does not model communication cost.

6.4 Other Related Work

The problem of general distributed credential storage has not been studied extensively in TM literature. QCM [12] and SD3 [13] are previous trust-management systems that consider distributed storage of credentials. The approach in QCM and SD3 assumes that issuers initially store all credentials and every query is answered by doing a form of backward search. This is impractical for many applications, including the examples in this paper.

The language RT_0 is closely related to SDSI 2.0 and Delegation Logic. See Section 2.1 for a comparison of RT_0 credentials with name definition certificates in SDSI 2.0 and Section 3.1 for review of existing work on chain discovery in SDSI. RT_0 can be viewed as a syntactically sugared version of a subset of Delegation Logic (DL). Through a type-1 RT_0 credential, the issuer can express the judgement that a subject has a certain attribute. A basic credential in DL has only an issuer and a statement. Although one can encode the subject and attribute together in a statement, DL lacks the explicit subject abstraction, which we desire for the following reasons. The explicit abstraction allows clear, concise representation of attribute-based delegation, *e.g.*, in the form of linked roles. As we have seen in this paper, the subject abstraction enables distributed storage and discovery of credentials. It also enables us to view attributes similarly to roles in Role-Based Access Control (RBAC) [19], and to use concepts similar to role activations to enable entities to make selective use of those roles. See [17] for more details.

7 Conclusions

We have introduced a simple Role-based Trust-management language RT_0 and a set-theoretic semantics for it. We have also introduced credential graphs as a searchable representation of credentials in RT_0 and have proven that reachability in credential graphs is sound and complete with respect to the semantics of RT_0. Based on credential graphs, we have given goal-directed algorithms to do credential chain discovery in RT_0 both when credential storage is centralized and when credential storage is distributed. Goal-directed algorithms provide better expected-case performance than do bottom-up algorithms, such as those developed by previous work in the context of SDSI. We have also introduced a type system for credential storage that guarantees distributed, well typed credential chains can be discovered by the distributed search algorithm. This type system can also be used to help improve the efficiency of search by guiding search in the right direction, making distributed chain discovery with large numbers of credentials feasible.

Acknowledgement

This work is supported by DARPA through SPAWAR contracts N66001-00-C-8015 and N66001-01-C-8005, and through AFRL/IF contract F30602-97-C-0336. It is also supported by DoD MURI "Semantics Consistency in Information Exchange" as ONR Grant N00014-97-1-0505. We thank the anonymous reviewers for their helpful comments.

A Proofs

A.1 Proof of Theorem 2

Proof. The proof is by induction on $\{\mathsf{rmem}^i\}_{i\in\mathcal{N}}$, whose limit is $\mathcal{S_C}$. We show that for each $i \in \mathcal{N}$, for any D and e_0, if $D \in \mathsf{expr}[\mathsf{rmem}^i](e_0)$, then $e_0 \xleftarrow{*} D$ exists in $G_{\mathcal{C}:\{e_0\}}$. The basis is trivial. For the step, fix any D and e_0 such that $D \in \mathsf{expr}[\mathsf{rmem}^{i+1}](e_0)$. There are four cases based on the structure of e_0. For cases 1–3, we show that $e_0 \xleftarrow{*} D$ exists in $G_\mathcal{C}$, which is a subgraph of $G_{\mathcal{C}:\{e_0\}}$.
case 1: e_0 is an entity, in which case it has to be D. We have $D \xleftarrow{*} D$ as the required path.
case 2: e_0 is a role $A.r$, in which case $D \in \mathsf{rmem}^{i+1}(A.r)$. By definition of rmem^{i+1} there exists $A.r \longleftarrow e \in \mathcal{C}$ such that $D \in \mathsf{expr}[\mathsf{rmem}^i](e)$. $G_\mathcal{C}$ has $A.r \leftarrow e$ by closure property 1. By induction hypothesis, $e \xleftarrow{*} D$ exists in $G_{\mathcal{C}:\{e\}}$, which equals $G_\mathcal{C}$, since e appears in a credential. Therefore, we have the required path $A.r \leftarrow e \xleftarrow{*} D$ in $G_\mathcal{C}$.
case 3: e_0 is a linked role $A.r_1.r_2$, in which case $D \in \mathsf{expr}[\mathsf{rmem}^{i+1}](A.r_1.r_2)$, which implies $D \in \mathsf{expr}[\mathsf{rmem}^{i+1}](B.r_2)$ for some $B \in \mathsf{expr}[\mathsf{rmem}^{i+1}](A.r_1)$. From the result proved in case 2, $B.r_2 \xleftarrow{*} D$ and $A.r_1 \xleftarrow{*} B$ exist in $G_\mathcal{C}$. Now by closure property 2, we have $A.r_1.r_2 \leftarrow B.r_2 \xleftarrow{*} D$.
case 4: e_0 is an intersection $f_1 \cap \cdots \cap f_k$, in which case $D \in \mathsf{expr}[\mathsf{rmem}^{i+1}](f_1 \cap \cdots \cap f_k)$, which implies $D \in \mathsf{expr}[\mathsf{rmem}^{i+1}](f_j)$ for each $j \in [1..k]$. From the results proved in the previous three cases, $f_j \xleftarrow{*} D$ exists in $G_\mathcal{C}$ for each $j \in [1..k]$. By closure property 3, $f_1 \cap \cdots \cap f_k \leftarrow D$ exists in $G_{\mathcal{C}:\{e_0\}}$ and is the required path. ∎

A.2 Proof of Theorem 3

Proof. We analyze the structure of the proof graph constructed by doing backward search.

Observation 1: Each node has $O(N)$ solutions. To become a solution of any node other than itself, an entity must appear alone as the body of a credential. There are $O(N)$ such entities.

Observation 2: The proof graph has $O(M)$ entity nodes, $O(M)$ linked-role nodes, $O(N)$ intersection nodes, and $O(NM)$ role nodes. Among these, credentials in $\mathcal{C}$ contribute $O(M)$ entity nodes, $O(M)$ linked-role nodes, $O(N)$ intersection node, and $O(M)$ role nodes. In addition, a role node $B.r_2$ is created whenever there is a node $A.r_1.r_2$ such that $A.r_1$ has B as a solution. Each of the $O(M)$ linked-role nodes generates $O(N)$ such role nodes (following observation 1).

Observation 3: Among all the role nodes, only $O(N)$ have any solution at all. For a role node to have a solution, it has to be defined by a credential in $\mathcal{C}$. There are $O(N)$ such roles.

Observation 4: Each role node $B.r_2$ has $O(N)$ outgoing edges of the form $A.r_1.r_2 \leftarrow B.r_2$. Each such edge corresponds to a role node $A.r_1$ with B as a solution. From observation 3, there are $O(N)$ such roles.

If we view monitors as special edges, the dominating factor in time complexity is the number of times that a solution passes through an edge. Each solution traverses an edge, $e_2 \leftarrow e_1$, at most once, since it is added to e_1 at most once. There are five classes of edges: (1) $O(N)$ credential edges, each corresponding to a credential in $\mathcal{C}$; (2) $O(M)$ linking monitors, each added on a role node $A.r_1$ while processing a linked-role node $A.r_1.r_2$; (3) $O(M)$ intersection monitors, each added on a node f_j while processing $f_1 \cap \cdots \cap f_k$; (4) $O(N^2)$ edges of the form $f_1 \cap \cdots \cap f_k \leftarrow D$ (the number $O(N^2)$ follows from observations 1 and 2); and (5) $O(NM)$ derived link edges, each of the form $A.r_1.r_2 \leftarrow B.r_2$, where B is a solution on $A.r_1$ (each of $O(M)$ linked roles generates $O(N)$ derived link edges, by observations 1 and 2).

There are $O(M)$ edges in the first three classes. Together with observation 1, this says the total cost incurred by them is $O(NM)$. Each of the $O(N^2)$ class-4 edges has cost 1, because a class-4 edge starts from an entity node, which has just one solution. Now consider the total cost incurred by class-5 edges. Because each derived link edge starts from a role node, this cost is bounded by the total cost incurred by all edges leaving role nodes. There are $O(N)$ role nodes that have solutions (observation 3), each has $O(N)$ solutions (observation 1) and $O(N)$ outgoing edges (observation 4), so the total cost for class-5 edges is $O(N^3)$. All together, the time complexity is $O(N^3 + NM)$.

Space complexity is bounded by the largest of the number of edges, $O(NM)$, the number of nodes, $O(NM)$, and the sum of the number of solutions on all nodes. There are four kinds of nodes: $O(M)$ entity nodes, each having 1 solution; $O(M)$ linked-role nodes, each having $O(N)$ solutions; $O(N)$ intersection nodes, each having $O(N)$ solutions; and $O(NM)$ role nodes, which together have $O(N^2)$ solutions (by observations 1 and 3). Therefore, the space complexity is $O(NM)$. ∎

A.3 Proof of Theorem 4

Proof. The proof is similar to that of Theorem 3. We analyze the structure of the proof graph constructed by the forward search algorithm.

Observation 1: Each node has $O(N)$ full solutions and $O(M)$ partial solutions. For a role to become a full solution to any node other than itself, it has to be defined by some credential; there are thus $O(N)$ full solutions. For each intersection $f_1 \cap \cdots \cap f_k$, $O(k)$ partial solutions are generated; therefore, the total number of partial solutions are $O(M)$.

Observation 2: The proof graph has $O(N)$ intersection nodes, $O(N)$ role nodes, $O(N)$ entity nodes, and $O(N^2)$ linked-role nodes. There are only $O(N)$ intersections in $\mathcal{C}$. The only way that a role node $A.r$ is added is through using a credential that defines $A.r$, so there are $O(N)$ such nodes. Except for the starting node D, the only way that an entity node B is added is though adding a role node $B.r_2$; thus, there are $O(N)$ entity nodes. A linked-role node $A.r_1.r_2$ is added when the role node $B.r_2$ is in the graph, and B has $A.r_1$ as a full solution. Each of the $O(N)$ role nodes generates $O(N)$ linked-role nodes (following observation 1).

Observation 3: The total number of solutions on all linked-role nodes are $O(NM)$. There are $O(N)$ linked roles that have any full solutions, because such a linked role has to appear by itself as the body of some credential in $\mathcal{C}$. The total number of solutions on these nodes is $O(NM)$, following observation 1. Each linked role that has no full solution gets one partial solution for each place it occurs in an intersection in $\mathcal{C}$. The total number of such occurrences for all linked-role nodes is $O(M)$, so the total number of solutions for these nodes is $O(M)$.

If we view monitors as special edges, the dominating factor in time complexity is the number of times that a solution passes through an edge. There are four kinds of edges: (1) $O(N)$ credential edges; (2) $O(N)$ linking monitors, each of which is added on an entity node B while processing $B.r_2$ (the number $O(N)$ follows from observation 2); (3) $O(N^2)$ edges of the form $f_1 \cap \cdots \cap f_k \leftarrow D$ (the number $O(N^2)$ follows from observation 2); and (4) $O(N^2)$ derived link edges, each of which has the form $A.r_1.r_2 \leftarrow B.r_2$, where $A.r_1$ is a full solution on B (each of the $O(N)$ role nodes generates $O(N)$ derived link edges, following observation 1 and 2). There are $O(N^2)$ edges in total. Together with observation 1 and the fact that $N = O(M)$, this says the total cost incurred by them is $O(N^2M)$).

The space complexity is bounded by the largest of the number of edges, $O(N^2)$, the number of nodes, $O(N^2)$, and the sum of the number of solutions on all nodes. Following observation 2, the total number of entity nodes, role nodes, and intersection nodes is $O(N)$; they have $O(NM)$

solutions. By observation 3, the total cost for linked-role nodes is also $O(NM)$. Therefore, the space complexity is $O(NM)$. ∎

Forward search from A_0 with the following set of credentials reaches the worst-case bounds.

$$\mathcal{C} = \left\{ \begin{array}{c} A_0.r \longleftarrow A_i,\ A_i.r \longleftarrow A_{i-1 \bmod n}.r,\ A_i.r' \longleftarrow A_i.r.r,\ A_i.r' \longleftarrow A_{i-1 \bmod n}.r', \\ A_i.r'' \longleftarrow A_i.r' \cap A_{i+1 \bmod n}.r' \cap \cdots \cap A_{i+n-2 \bmod n}.r' \end{array} \middle| \, 0 \le i < n \right\}$$

A.4 Proof of Theorem 5

Proof. $\mathcal{C}'$ has $O(NL)$ credentials, so from Theorem 3, the time complexity is $O(N^3L^3)$. However, a more detailed analysis yields a tighter bound. Breaking each $A.r \longleftarrow B.r_1.r_2.\ldots.r_k$ into $\{A.r \longleftarrow A.r'_{k-1}.r_k,\ A.r'_{k-1} \longleftarrow A.r'_{k-2}.r_{k-1},\ \ldots,\ A.r'_2 \longleftarrow A.r'_1.r_2,\ A.r'_1 \longleftarrow B.r_1\}$ does not introduce new potential solutions; there are still $O(N)$ of them. Now, consider the five classes of edges listed in the proof of Theorem 3. There are $O(NL)$ credential edges and linking monitors, no class-3 or class-4 edges, and $O(N^2L)$ derived link edges (each of the $O(NL)$ linked roles generates $O(N)$ such edges). Therefore, the time complexity is $O(N^3L)$. ∎

A.5 Proof of Theorem 8

Proof. We prove the three propositions simultaneously by considering edges in $\langle e_2 \twoheadleftarrow e_1 \rangle$ and using induction over the steps of the construction of $E^i_{\mathcal{C}:Q}$ in Definition 2.

Base Case: the chain $\langle e_2 \twoheadleftarrow e_1 \rangle$ contains only edges in $E^0_{\mathcal{C}:Q}$, *i.e.*, credential edges; in which case $\langle e_2 \twoheadleftarrow e_1 \rangle = e_2 \overset{*}{\leftarrow} e_1$.

prop 1: $\langle e_2 \twoheadleftarrow e_1 \rangle$ is backward traversable. We need to prove that if e_2 is b-activated in PG_R, then $\mathcal{C}(R) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$. We use an inner induction on the length of $e_2 \overset{*}{\leftarrow} e_1$. The inner base case, in which $e_1 = e_2$, is trivial. For the step, we decompose $e_2 \overset{*}{\leftarrow} e_1$ into $e_2 \leftarrow e' \overset{*}{\leftarrow} e_1$. Then there is a credential $e_2 \longleftarrow e'$ that is stored with its issuer, and $\mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle = e_2 \longleftarrow e' \cup \mathcal{C}\langle e' \twoheadleftarrow e_1 \rangle$. The existence of the credential $e_2 \longleftarrow e'$ implies that e_2 is a role; backward processing of e_2 discovers the credential $e_2 \longleftarrow e'$ (L14), and adds the edge, which results in e' being b-activated (L10 via L15). Inner induction hypothesis says that $\mathcal{C}(R) \supseteq \mathcal{C}\langle e' \twoheadleftarrow e_1 \rangle$, and so this proposition follows.

prop 2: $e_2 \overset{*}{\leftarrow} e_1$ is forward traversable. We need to prove that if e_1 is f-activated in PG_R, then $\mathcal{C}(R) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$. We use an inner induction on the length of $e_2 \overset{*}{\leftarrow} e_1$, and for the step, decompose $e_2 \overset{*}{\leftarrow} e_1$ into $e_2 \overset{*}{\leftarrow} e' \leftarrow e_1$. Then there is a credential $e' \longleftarrow e_1$ that is stored with each entity in $base(e_1)$. Forward processing e_1 discovers the credential, and adds the edge, which results in e' being f-activated (L34). Inner induction hypothesis says that $\mathcal{C}(R) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e' \rangle$, and so the proposition follows.

prop 3: $e_2 \overset{*}{\leftarrow} e_1$ is confluent. We need to prove that if e_1 is f-activated in R_1 and e_2 is b-activated in R_2, then $\mathcal{C}(R_2) \cup \mathcal{C}(R_1) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$. By definition, $e_2 \overset{*}{\leftarrow} e_1$ can be decomposed into $e_2 \overset{*}{\leftarrow} e'' \leftarrow e' \overset{*}{\leftarrow} e_1$, with $e' \overset{*}{\leftarrow} e_1$ forward traversable, $e_2 \overset{*}{\leftarrow} e''$ backward traversable, and $e'' \leftarrow e'$ confluent. A confluent edge in $E^0_{\mathcal{C}:Q}$ is either backward or forward traversable. If $e'' \leftarrow e'$ is backward traversable, so is $e_2 \overset{*}{\leftarrow} e'$. The two propositions above say that $\mathcal{C}(R_2) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e' \rangle$ and $\mathcal{C}(R_1) \supseteq \mathcal{C}\langle e' \twoheadleftarrow e_1 \rangle$. And so $\mathcal{C}(R_2) \cup \mathcal{C}(R_1) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle$. Similar arguments cover the case that $e'' \leftarrow e'$ is forward traversable, and so the proposition follows.

Induction Step: We consider the case when $\langle e_2 \twoheadleftarrow e_1 \rangle \subseteq E^{i+1}_{\mathcal{C}:Q}$ but $\langle e_2 \twoheadleftarrow e_1 \rangle \not\subseteq E^i_{\mathcal{C}:Q}$.

prop 1: Let $e'' \leftarrow e'$ be the (unique) edge in $E^{i+1}_{\mathcal{C}:Q} - E^i_{\mathcal{C}:Q}$. Then $e'' \leftarrow e'$ is a derived edge, and it is on the path $e_2 \overset{*}{\leftarrow} e_1$; therefore, $e_2 \overset{*}{\leftarrow} e_1$ can be decomposed into $e_2 \overset{*}{\leftarrow} e'' \leftarrow e' \overset{*}{\leftarrow} e_1$, in which chains $\langle e_2 \twoheadleftarrow e'' \rangle$ and $\langle e' \twoheadleftarrow e_1 \rangle$ are in $E^i_{\mathcal{C}:Q}$, and so $\mathcal{C}\langle e_2 \twoheadleftarrow e_1 \rangle = \mathcal{C}\langle e_2 \twoheadleftarrow e'' \rangle \cup \mathcal{C}\langle e'' \twoheadleftarrow e_1 \rangle$. Because $\langle e_2 \twoheadleftarrow e_1 \rangle$ is backward traversable, so are $\langle e_2 \twoheadleftarrow e'' \rangle$, $e'' \leftarrow e'$, and $\langle e' \twoheadleftarrow e_1 \rangle$. When e_2 is b-activated, by induction hypothesis, $\mathcal{C}(R) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e'' \rangle$; furthermore, from Theorem 7, e'' is b-activated. We still need to prove $\mathcal{C}(R) \supseteq \mathcal{C}\langle e'' \twoheadleftarrow e_1 \rangle$. Based on how $e'' \leftarrow e'$ is introduced, there are two cases.

One, $e'' \leftarrow e'$ is a derived link edge of the form $A.r_1.r_2 \leftarrow B.r_2$, and $\langle A.r_1 \twoheadleftarrow B \rangle$ is in $E^i_{\mathcal{C}:Q}$. Then we have $\mathcal{C}\langle e'' \twoheadleftarrow e_1 \rangle = \mathcal{C}\langle A.r_1 \twoheadleftarrow B \rangle \cup \mathcal{C}\langle B.r_2 \twoheadleftarrow e_1 \rangle$. Since $A.r_1.r_2$ is b-activated, it is backward processed, and so $A.r_1$ in b-activated (L18). By induction hypothesis, $\mathcal{C}(R) \supseteq \mathcal{C}\langle A.r_1 \twoheadleftarrow B \rangle$; from Theorem 7, $A.r_1$ has B as a backward solution, which results in $B.r_2$ being created and b-activated (L20). By induction hypothesis, $\mathcal{C}(R) \supseteq \mathcal{C}\langle B.r_2 \twoheadleftarrow e_1 \rangle$. Therefore, the proposition holds.

Two, $e'' \leftarrow e'$ is a derived intersection edge, in which case it has the form $f_1 \cap \cdots \cap f_k \leftarrow D$, and for each $j \in [1..k]$, $\langle f_j \twoheadleftarrow D \rangle$ are in $E^i_{\mathcal{C}:Q}$. In addition, $e' = e_1 = D$, and so $\mathcal{C}\langle e'' \twoheadleftarrow e_1 \rangle = \bigcup_{1 \le j \le k} \mathcal{C}\langle f_j \twoheadleftarrow D \rangle$. Backward processing $e'' = f_1 \cap \cdots \cap f_k$ results in each of f_j being created and b-activated (L25). By the definition of $f_1 \cap \cdots \cap f_k \leftarrow D$'s backward traversability, there is one f_ℓ with $\langle f_\ell \twoheadleftarrow D \rangle$ backward traversable. By induction hypothesis, $\mathcal{C}(R) \supseteq \mathcal{C}\langle f_\ell \twoheadleftarrow D \rangle$; from Theorem 7, f_ℓ has D as a backward solution, and so D will reach the intersection monitor for $f_1 \cap \cdots \cap f_k$. The algorithm then creates D and f-activates it (L38). Since each chain $\langle f_j \twoheadleftarrow D \rangle$ is in $E^i_{\mathcal{C}:Q}$ and is confluent, by induction hypothesis 3, $\mathcal{C}(R) \cup \mathcal{C}(R) \supseteq \mathcal{C}\langle f_j \twoheadleftarrow D \rangle$. Therefore, the proposition follows.

prop 2: When $\langle e_2 \twoheadleftarrow e_1 \rangle$ is forward traversable, the arguments are similar to those for proposition 1.

prop 3: When $\langle e_2 \twoheadleftarrow e_1 \rangle$ is confluent. By definition, the path $e_2 \overset{*}{\leftarrow} e_1$ can be decomposed into $e_2 \overset{*}{\leftarrow} e'' \leftarrow e' \overset{*}{\leftarrow} e_1$, in which $\langle e' \twoheadleftarrow e_1 \rangle$ is forward traversable, $\langle e_2 \twoheadleftarrow e'' \rangle$ backward traversable, and $e'' \leftarrow e'$ confluent. The two propositions above say that $\mathcal{C}(R_2) \supseteq \mathcal{C}\langle e_2 \twoheadleftarrow e'' \rangle$ and $\mathcal{C}(R_1) \supseteq \mathcal{C}\langle e' \twoheadleftarrow e_1 \rangle$. From Theorem 7, e'' is b-activated in R_2, and e' is f-activated in R_1. The edge $e'' \leftarrow e'$ has the following three cases.

case a: $e'' \leftarrow e'$ is a credential edge, in which case $e'' \leftarrow e'$ is either backward or forward traversable, and so the credential $e'' \longleftarrow e'$ is in $\mathcal{C}(R_1)$ or $\mathcal{C}(R_2)$.

case b: $e'' \leftarrow e'$ is a derived link edge $A.r_1.r_2 \leftarrow B.r_2$, in which case $\langle A.r_1 \twoheadleftarrow B \rangle$ is in $E^i_{\mathcal{C}:Q}$ and is confluent. Since $e'' = A.r_1.r_2$ is b-activated in R_2, so is $A.r_1$ (L18). Since $e' = B.r_2$ is f-activated in R_1, so is B (L32). By induction hypothesis, $\mathcal{C}(R_2) \cup \mathcal{C}(R_1) \supseteq \mathcal{C}\langle A.r_1 \twoheadleftarrow B \rangle$. Therefore, the proposition follows.

case c: $e'' \leftarrow e'$ is a derived intersection edge $f_1 \cap \cdots \cap f_k \leftarrow D$, in which case for each $j \in [1..k]$, there is a confluent chain $\langle f_j \twoheadleftarrow D \rangle$ in $E^i_{\mathcal{C}:Q}$. Backward processing $e'' = f_1 \cap \cdots \cap f_k$ b-activates f_j for each $j \in [1..k]$ (L25). By induction hypothesis, for each $j \in [1..k]$, $\mathcal{C}(R_2) \cup \mathcal{C}(R_1) \supseteq \mathcal{C}\langle f_j \twoheadleftarrow D \rangle$. Therefore, the proposition follows. ∎

A.6 Proof of Theorem 9

Proof. By Definition 4, $\langle e \twoheadleftarrow e_1 \rangle$ has the same traversability as $e \overset{*}{\leftarrow} e_1$, so we prove the three propositions for $e \overset{*}{\leftarrow} e_1$. We do this simultaneously by using an induction over the steps of the construction of $E^i_{\mathcal{C}:Q}$ given in Definition 2.

Base Case: Here, the path $e \overset{*}{\leftarrow} e_1$ contains only edges in $E^0_{\mathcal{C}:Q}$, *i.e.*, credential edges. We use a second, inner induction on the length of $e \overset{*}{\leftarrow} e_1$. The inner base case, in which $e = e_1$, is trivial. For the step, we decompose $e \overset{*}{\leftarrow} e_1$ into $e \leftarrow e_2 \overset{*}{\leftarrow} e_1$. Because $e \leftarrow e_2$ is in $E^0_{\mathcal{C}:Q}$, there is a well typed credential of the form $e \longleftarrow e_2$.

When e is issuer-traces-all, the credential $e \longleftarrow e_2$ is stored with its issuer, and e_2 is also issuer-traces-all, by well-typedness of $e \longleftarrow e_2$ (Definition 7). By induction hypothesis, $e_2 \stackrel{*}{\leftarrow} e_1$ is backward traversable. Therefore, $e \leftarrow e_2 \stackrel{*}{\leftarrow} e_1$ is backward traversable, proving Proposition 2. Thus, it is trivially confluent, proving Proposition 1. When e is subject-traces-all, a similar argument proves propositions 1 and 3.

When e is neither issuer-traces-all nor subject-traces-all, it is weakly well typed, *i.e.*, issuer-traces-def. Therefore the edge $e \leftarrow e_2$ is backward traversable. By induction hypothesis, $e_2 \stackrel{*}{\leftarrow} e_1$ is confluent, so either $e_2 \stackrel{*}{\leftarrow} e_1$ is empty, in which case $e \stackrel{*}{\leftarrow} e_1$ is just $e \leftarrow e_2$ and is therefore confluent, or $e_2 \stackrel{*}{\leftarrow} e_1$ can be decomposed into $e_2 \stackrel{*}{\leftarrow} e'' \leftarrow e' \stackrel{*}{\leftarrow} e_1$, where $e' \stackrel{*}{\leftarrow} e_1$ is forward traversable, $e_2 \stackrel{*}{\leftarrow} e''$ is backward traversable, and $e'' \leftarrow e'$ is confluent. Because $e \leftarrow e_2 \stackrel{*}{\leftarrow} e''$ is also backward traversable, proposition 1 follows.

Induction Step: We also use an inner induction on the length of $e \stackrel{*}{\leftarrow} e_1$, in $E^{i+1}_{\mathcal{C}:Q}$. Again the basis is trivial. For the inner step, we decompose the path $e \stackrel{*}{\leftarrow} e_1$ into $e \leftarrow e_2 \stackrel{*}{\leftarrow} e_1$. There are three cases for $e \leftarrow e_2$.

case 1: When $e \leftarrow e_2$ is a credential edge, the argument is identical to the inner step of the outer base case.

case 2: When $e \leftarrow e_2$ is a derived link edge $A.r_1.r_2 \leftarrow B.r_2$, there is a path from B to $A.r_1$ in $E^{i}_{\mathcal{C}:Q}$, by construction of $E^{i+1}_{\mathcal{C}:Q}$. When $A.r_1.r_2$ is issuer-traces-all, so are r_1 and r_2, by Definition 6. The outer induction hypothesis says that $A.r_1 \stackrel{*}{\leftarrow} B$ is backward traversable. Consequently, $A.r_1.r_2 \leftarrow B.r_2$ is backward traversable. The inner induction hypothesis, together with the fact that r_2 is issuer-traces-all, tell us that $B.r_2 \stackrel{*}{\leftarrow} e_1$ is backward traversable. Propositions 1 and 2 now follow. When $A.r_1.r_2$ is subject-traces-all, so are r_1 and r_2. Propositions 1 and 3 can be shown by direct analogy with the issuer-traces-all case.

When $A.r_1.r_2$ is neither issuer-traces-all nor subject-traces-all, it is weakly well typed. There are two subcases corresponding to the two cases of the weakly well typed rule for $A.r_1.r_2$.

case 2a: r_1 is issuer-traces-all and r_2 is well typed. The inner induction hypothesis tells us that $B.r_2 \stackrel{*}{\leftarrow} e_1$ is confluent. The fact that r_1 is issuer-traces-all, together with the outer induction hypothesis, tells that $A.r_1 \stackrel{*}{\leftarrow} B$ is backward traversable, which in turn tells us that $A.r_1.r_2 \leftarrow B.r_2$ is backward traversable. Confluence of $A.r_1.r_2 \leftarrow B.r_2 \stackrel{*}{\leftarrow} e_1$ now follows by using the definition of path confluence.

case 2b: r_1 is well typed and r_2 is subject-traces-all. The latter, together with the inner induction hypothesis, tells us that $B.r_2 \stackrel{*}{\leftarrow} e_1$ is forward traversable. The outer induction hypothesis tells that $A.r_1 \stackrel{*}{\leftarrow} B$ is confluent, so $A.r_1.r_2 \leftarrow B.r_2$ is, too. Confluence of $A.r_1.r_2 \leftarrow B.r_2 \stackrel{*}{\leftarrow} e_1$ now follows by using the definition of path confluence.

case 3: When $e \leftarrow e_2$ is a derived intersection edge $f_1 \cap \cdots \cap f_k \leftarrow D$, $e_1 = D$ (since entities have no incoming edges), and for each $j \in [1..k]$, there is $f_j \stackrel{*}{\leftarrow} D$ in $E^{i}_{\mathcal{C}:Q}$.

When $f_1 \cap \cdots \cap f_k$ is issuer-traces-all, there is an ℓ such that f_ℓ is issuer-traces-all and the other f_j are well typed. It follows from the outer induction hypothesis that $f_\ell \stackrel{*}{\leftarrow} D$ is backward traversable and that for each $j \in [1..k]$, $f_j \stackrel{*}{\leftarrow} D$ is confluent. Definition 4 says that $f_1 \cap \cdots \cap f_k \leftarrow D$ is backward traversable. Since $e \stackrel{*}{\leftarrow} e_1 = f_1 \cap \cdots \cap f_k \leftarrow D$, it is backward traversable, too.

When $f_1 \cap \cdots \cap f_k$ is subject-traces-all, it follows that $f_1 \cap \cdots \cap f_k \stackrel{*}{\leftarrow} D$ is forward traversable by an argument that is almost identical to the previous case.

When $f_1 \cap \cdots \cap f_k$ is issuer-traces-def, we know that each f_j is well typed. The outer induction hypothesis then tells us that each $f_j \overset{*}{\leftarrow} B$ is confluent. Consequently, $f_1 \cap \cdots \cap f_k \leftarrow B$ is also confluent. This means that $e \overset{*}{\leftarrow} e_1$ is confluent, as required to complete the proof. ∎

A.7 Proof of Theorem 9 with Extended Type System

Proof. The proof for Theorem 9 in the previous section applies almost without change. The only change is in the step of the inner induction.

When the path $e \overset{*}{\leftarrow} e_1$ in $E^0_{\mathcal{C}:Q}$ is decomposed into $e \leftarrow e_2 \overset{*}{\leftarrow} e_1$, and e is weakly well typed, there is a case other than e being issuer-traces-def; e may be issuer-traces-rule and subject-traces-fact. We need to prove that $e \overset{*}{\leftarrow} e_1$ is confluent. When e_2 is not an entity, the edge $e \leftarrow e_2$ is backward traversable; this follows essentially as it does in the issuer-traces-def case. When e_2 is an entity, $e_1 = e_2$ is also an entity, since an entity node has no incoming edge. Thus the path $e \overset{*}{\leftarrow} e_1$ consists of one credential stored with its subject, which is clearly confluent. ∎

References

[1] Tuomas Aura. Fast access control decisions from delegation certificate databases. In *Proceedings of 3rd Australasian Conference on Information Security and Privacy (ACISP '98)*, volume 1438 of *Lecture Notes in Computer Science*, pages 284–295. Springer, 1998.

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

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

[4] 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.

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

[6] 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.

[7] Yassir Elley, Anne Anderson, Steve Hanna, Sean Mullan, Radia Perlman, and Seth Proctor. Building certificate paths: Forward vs. reverse. In *Proceedings of the 2001 Network and Distributed System Security Symposium (NDSS'01)*, pages 153–160. Internet Society, February 2001.

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

[9] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. Simple public key certificates. Internet Draft (work in progress), July 1999. http://world.std.com/~cme/spki.txt.

[10] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.

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

[12] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. *Software: Practice & Experience*, 30(15):1609–1640, September 2000.

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

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

[15] Ninghui Li. Local names in SPKI/SDSI. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 2–15. IEEE Computer Society Press, July 2000.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.