Distributed Credential Chain Discovery in Trust Management with Parameterized Roles and Constraints (Short Paper)

Ziqing Mao[1], Ninghui Li[1], and William H. Winsborough[2]

[1] CERIAS and Department of Computer Science, Purdue University
{zmao,ninghui}@cs.purdue.edu
[2] Department of Computer Science, University of Texas at San Antonio
{wwinsborough}@acm.org

Abstract. Trust management (TM) is an approach to access control in decentralized distributed systems with access control decisions based on statements made by multiple principals. Li et al. developed the RT family of Role-Based Trust-management languages, which combine the strengths of Role-Based Access Control and TM systems. We present a distributed credential chain discovery algorithm for RT_1^C, a language in the RT family that has parameterized roles and constraints. Our algorithm is a combination of the logic-programming style top-down query evaluation with tabling and a goal-directed version of the deductive database style bottom-up evaluation. Our algorithm uses hints provided through the storage types to determine whether to use a top-down or bottom-up strategy for a particular part of the proof; this enables the algorithm to touch only those credentials that are related to the query, which are likely to be a small fraction of all the credentials in the system.

1 Introduction

In [1], Blaze, Feigenbaum, and Lacy coined the term "trust management" to group together some principles dealing with access control in decentralized distributed systems. In the TM approach, access control decisions are based on the attributes (rather than the identity) of the requester, such as citizenship, credit status, date of birth, employment, group membership, security clearance, etc. These attributes need to be certified: they are documented by digitally-signed credentials issued by appropriate authorities, which may have their own attributes documented in other credentials. When one requests a resource from a server, the access is granted if the requester's attributes in its credentials satisfy the server's policy. TM systems allow the authority to certify attributes to be delegated. Like attributes themselves, such delegation relationships are documented in credentials. For example, a university can issue a credential to delegate to its registrar the authority to certify who are students of the university. The process of making an access control decision involves finding a chain of credentials that together prove that the requester satisfies the server's policy. 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*.

In a series of papers [12, 11, 10], Li et al. developed the RT family of Role-Based Trust-management languages, which combine the strengths of Role-Based Access Control (RBAC) [16] and TM systems. Two central concepts in RT are *principals* and *roles*. Each principal represents a uniquely identified entity in the system. A role is designated by a principal and a *role term*. For example, $\mathrm{HospB.physician}$ is the $\mathrm{physician}$ role defined by principal HospB and can be read as HospB's $\mathrm{physician}$ role. In RT_0, the most basic language in the RT family, each role term is a string. Li et al. [12] introduced an approach for doing distributed credential chain discovery in RT_0. In credential chain discovery, one needs to determine whether a requester has the attributes that satisfy the policy. One approach is to use *backward search*, which starts with the policies that govern the requested resource and tries to enumerate all principals that are entitled to access the resource. One difficulty of this approach is that because of recursive dependency in policies, the search may never terminate. Clarke et al. [4] proposed an algorithm that addresses the problem by doing a full-scale *forward search*, which tries to compute all facts entailed by all the credentials and policies in the system. These approaches have two drawbacks: First, using either forward or backward search alone, one may evaluate a large number of credentials unrelated to the query. Second, when credentials are stored in a distributed manner, one may not know the existence of some relevant credentials. The approach in [12] addresses these problems by using a goal-directed chain discovery algorithm that combines goal-directed back search and goal-directed forward search.

In [11, 10], a number of other components of RT were introduced. In particular, RT_1 adds parameterized roles to RT_0. Parameterized roles can represent attributes that have fields. For example, if HospB has a policy that allows the primary care physician (pcp) of a patient to access the patient's medical record, then HospB needs to define the pcp role. Without parameterized roles, HospB needs to define a pcp role for each patient and to grant access to each of these roles individually. In RT_1, one can parameterize the role $\mathrm{HospB.pcp}$ by patient id, and then use only one statement to express the policy. RT_1^C enhances RT_1 with constraints. This enables one to succinctly express permissions regarding structured resources and potentially unbounded domains. For example, using one statement, one can grant the permission to connect to any port over 1024 at any host in the domain `abc.dom`. Clearly these are essential capabilities in a real-world policy language.

While introducing parameterized roles and constraints greatly increases the expressive power of the RT family, credential chain discovery was also made significantly more challenging. In this paper, we present a distributed credential chain discovery algorithm for RT_1^C. Our algorithm is a novel combination of goal-directed backward search with tabling and goal-directed forward search, using a storage typing system and a mechanism for communicating results between the two search directions and managing search in the two directions. We describe this algorithm in detail; in our specification of the algorithm, we state logical invariants that ensure correctness.

The rest of this paper is organized as follows. Related work is discussed in Section 2. We give a detailed example scenario in Section 3. In Section 4, we describe the syntax and semantics of the RT_1^C language. Distributed credential chain discovery algorithms are given in Section 5. We conclude in Section 6.

2 Related Work

Clarke et al. [4] gave an algorithm for credential chain discovery in SPKI/SDSI 2.0 [5]. Their algorithm views discovery as a term-rewriting problem. Each certificate is viewed as a rewriting rule. Determining whether there is a credential chain that proves a role expression e has a member D is equivalent to whether there is a way to rewrite e into D. In order to avoid potential nontermination caused by recursive definitions, the algorithm in [4] computes a closure the member-sets of all roles in $\mathcal{C}$. 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. The algorithm in [4] also requires that evaluation begin by collecting all credentials in the system at a single location, where the evaluation will be carried out. This is a common problem with many evaluation techniques. In an open system, it will typically be the case that a large number of credentials have nothing to do with the current query. Evaluation methods should not require these irrelevant credentials to be collected. However, because there are no restrictions on the delegation of authority that credentials can specify, there is no simple means of determining which credentials are relevant without examining the chains and partial chains in which they participate. This is the principle our approach uses to avoid collecting irrelevant credentials.

Jha and Reps [7] pointed out that SDSI string rewriting systems correspond exactly to the class of string rewriting systems modeled using push-down systems [2], and therefore, one could use techniques for model checking pushdown systems to do credential chain discovery. This approach, however, does not extend to parameters and constraints.

Query Certificate Managers (QCM) [6] and Secure Dynamically Distributed Datalog (SD3) [8] also 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.

Li et al. [12] gave a distributed credential chain discovery algorithm for RT_0. Extending the algorithm in [12] to deal with parameterized roles and constraints turns out to be quite challenging. One can compare RT_0 to a propositional language, and RT_1^C to a first-order language.

As RT languages have a logic programming semantics, chain discovery in RT_1^C is closely related to deduction in logic programming and deductive databases. Backward search is top-down evaluation, which is used in Prolog engines; and forward search is similar to bottom-up evaluation, which is used in deductive databases. Issues such as tabling and goal-directed evaluation have been extensively studied. For example, top-down evaluation with tabling is studied in [3], and goal-directed bottom-up evaluation is studied in [13]. The uniqueness of our problem lies in the fact that it dictates a combination of top-down evaluation and bottom-up evaluation, because of the distributed storage of credentials. The search algorithm needs to be able to manage searches in both directions and to pass solutions from the search in one direction to the search in the other direction. Also, as RT_1^C has constraints; the search algorithm needs to incor-

porate ideas from the evaluation algorithms for constraint datalog (e.g., [17]). On the other hand, our problem is simpler than the general problem in that our algorithm only needs to handle four types of logical rules corresponding to the four types of statements in RT.

3 An Example

In this section, we describe an example scenario we will use throughout this paper to illustrate credentials and policy statements in RT and the distributed credential chain discovery process. This example is given in Figure 1 and explained below.

DC is a data center that maintains medical data about patients. The data maintained by it includes patient's personal information (such as name and birthdate), contact info, as well as other medical data such as test results and images. These data are labeled with category information, and the category information is organized in a hierarchy. Some sample categories are shown in the Figure 1(a). There are 3 categories at the top level: person, contact, and medical; each contains subcategories. For example, one's blood test result will be labeled with the category 'medical.testresult.blood' and one's email address will be labeled with the category 'contact.online.email'.

In the discussions below, we distinguish between policy statements and credentials. Policy statements are issued by DC and used by DC locally, thus they do not need to digitally signed. On the other hand, credentials are digitally signed, and DC needs to verify the signatures before accepting them. Other than the above difference, policy statements and credentials can be handled in exactly the same way in the chain discovery process. Note that credentials support the full generality of policy statements, and typically must be collected from distributed storage during chain discovery.

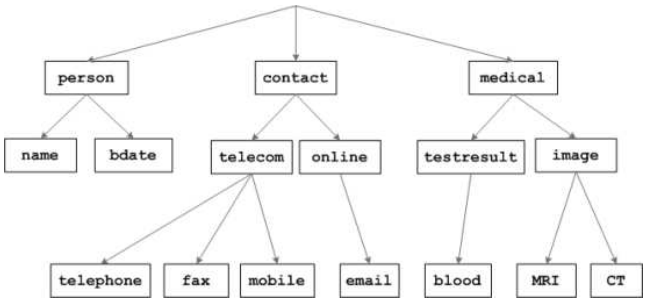
Policies DC's policy about accessing the data includes the following two rules:

- The primary care physician (PCP) of a patient has access to all information about a patient.
- The PCP of a patient is allowed to delegate access to medical info to another physician in an affiliated clinic or hospital.

These two rules are encoded using policy statements (p1) and (p2) in Figure 1(c). The statement `(p1)` [DC.access(pname=$?x$, data=$?y$)$\longleftarrow$DC.pcp(pname=$?x$)] states that any principal that is the PCP of a patient X can access any data about the patient X. In the statement $?x$ and $?y$ are two variables. DC.access(pname=$?x$, data=$?y$) is a parameterized role. Note that the same variable $?x$ appears both in the head (the part to the left of$\longleftarrow$) and the body (the part to the right of$\longleftarrow$).

The statement (p2) [DC.access(pname=$?x$, data=$?y$) $\longleftarrow$ DC.delAcc(pname=$?x$, data=$?y$) $\cap$ DC.physician ; $?y \preceq \langle$medical$\rangle$] states that if a principal is being delegated access to certain medical data, and is a physician, then the principal is allowed to access the data. The symbol $\cap$ denotes set intersection, if one views each role as the set of principals who are members of the role; it can also be equivalently viewed as a logical AND. Note that (p2) includes a constraint $?y \preceq \langle$medical$\rangle$, which means that $?y$ must be a subcategory of medical. This syntax for constraints will be explained in Section 4.

(a) The Category Hierarchy for the Patient Data



(b) Principals

DC	a data center that maintains medical data
ClinicA	a clinic that is affiliated with DC
HospB	a hospital that is affiliated with DC
Alice	a physician at ClinicA and the PCP of the patient 'Paul'
Bob	a physician at HospB, and is referred by Alice to access image data of 'Paul'

(c) Policy Statements and Credentials

label	statement	stored by
(p1)	DC.access(pname=$?x$, data=$?y$)$\longleftarrow$ DC.pcp(pname=$?x$). where pname stands for "patient name"	DC
(p2)	DC.access(pname=$?x$, data=$?y$)$\longleftarrow$ DC.delAcc(pname=$?x$, data=$?y$) $\cap$ DC.physician ; $?y \preceq \langle$medical$\rangle$.	DC
(p3)	DC.delAcc(pname=$?x$, data=$?y$)$\longleftarrow$ DC.pcp(pname=$?x$).refAcc(pname=$?x$, data=$?y$).	DC
(p4)	DC.pcp(pname=$?x$)$\longleftarrow$ DC.affil.pcp(pname=$?x$).	DC
(p5)	DC.physician$\longleftarrow$ DC.affil.physician.	DC
(c1)	DC.affil$\longleftarrow$ ClinicA.	ClinicA
(c2)	DC.affil$\longleftarrow$ HospB.	HospB
(c3)	ClinicA.pcp(pname=$?x$)$\longleftarrow$ Alice ; $?x =$'Paul'.	ClinicA
(c4)	HospB.physician$\longleftarrow$ Bob.	Bob
(c5)	Alice.refAcc(pname=$?x$, data=$?y$)$\longleftarrow$ Bob ; $?x =$ 'Paul' $\wedge$ $?y \preceq \langle$medical.image$\rangle$.	Bob

(d) The Inference Process:

label	conclusion	using
(r1)	DC.pcp(pname=$?x$) ; $?x =$'Paul' $\twoheadleftarrow$ Alice	(p4), (c1), (c3)
(r2)	DC.physician $\twoheadleftarrow$ Bob.	(p5), (c2), (c4)
(r3)	DC.delAcc(pname=$?x$, data=$?y$) ; $?x =$ 'Paul' $\wedge$ $?y \preceq \langle$medical.image$\rangle$ $\twoheadleftarrow$ Bob.	(p3), (r1), (c5)
(r4)	DC.access(pname=$?x$, data=$?y$) ; $?x =$ 'Paul' $\wedge$ $?y \preceq \langle$medical.image$\rangle$ $\twoheadleftarrow$ Bob.	(p2), (r3), (r2)

Fig. 1. A Running Example. This example is explained in detail Section 3.

One cannot use (p2) to gain access to data other than those under the medical category; for example, even if a physician is being delegated access to the contact information by the PCP, the physician still cannot use this rule to gain access to the information.

Policies (p1) and (p2) refer to the three roles: DC.delAcc($\cdots$), DC.pcp($\cdots$), and DC.physician. They are defined in (p3), (p4), and (p5), respectively. The policy (p3) states that one can be delegated access to a patient's data by the PCP of the patient; (p4) states that DC delegates the authority to certify the PCP relationship to members of the role DC.affil; and (p5) is a similar delegation about physicians.

Credentials and Inferences The data center DC may have many affiliated clinics and hospitals, each of which may have hundreds of physicians and thousands of patients, and there may be even more referring relationships. Therefore, in the whole system there may be millions of credentials. In this example, we consider only the five credentials in Figure 1(c).

Credential (c1) is issued by DC to ClinicA, and asserts that ClinicA is affiliated with DC. Credential (c3) is issued by ClinicA and asserts that Alice the PCP of the patient who has patient name[3] 'Paul'. From these two credentials and the policy (p4), one can infer (r1): Alice is a member of the constrained role DC.pcp(pname=?x); ?x ='Paul'. (In Figure 1(c), this is denoted by the syntax DC.pcp(pname=?x); ?x ='Paul' $\twoheadleftarrow$ Alice.) Similarly, from credentials (c2) and (c4), together with the policy (p5), one can infer (r2): Bob is a member of the role DC.physician.

Credential (c5) is issued by Alice when Alice wants Bob to look at the medical image date of patient 'Paul', maybe for a second opinion. From (p3), (r1), and (c5), one can infer (r3): Bob is a member of the role DC.delAcc(pname=?x, data=?y); ?x = 'Paul' $\wedge$?$y \preceq \langle$medical.image$\rangle$.

Finally, using (p2), (r3), and (r2), one can infer that Bob is able to get access to the medical image data of the patient 'Paul'. For example, if Bob requests to access the MRI image, then the access should be allowed.

Credential Storage and Discovery The first question that we need to address to enable the above access is: Suppose that DC maintains all credentials that are issued by everyone, when Bob requests access to the MRI image of patient 'Paul', how can one make the authorization decision efficiently? We point out that there may be tens of thousands of patients in the system, most of which are unrelated to the above access query; therefore, even if an algorithm runs in time linear in the total number of credentials, it is still not efficient enough. We need an algorithm that touches only the small fraction of credentials that are related to the query.

Furthermore, it is unreasonable to have DC maintain all credentials. For example, credentials (c3), (c4), and (c5) do not even mention DC. It is illogical to have DC store these credentials. The second question that we need to address is then how to find these credentials that are needed. For example, credential (c4) is issued by HospB and certify that Bob is a physician with HospB. Intuitively, it should be stored either by HospB or Bob. When we say a principal *stores* a credential, it means that we can

[3] In practice, patient records are more likely to be identified with unique patient ids, rather than names. We use patient names here to make the presentation smoother.

find the credential once we know the principal. Some system, such as a directory, may actually house the credential on the principal's behalf. We require that one can find the directory's address once knowing the principal. One approach to do this is to require the representation of a principal to include both the public key and the directory server address. See [15] for more discussions on this.

In [12], a storage type system and a notion of well-typed credentials were presented to address these problems. They guarantee that credential chains can be discovered even when credentials are stored in a distributed manner. The types also guide search in the right direction, avoiding huge fan-outs. See [12] for description of the storage type system.

4 An Overview of RT_1^C

Constraints RT_1^C uses constraints to support finite expression of authorizations over infinite or unbounded domains, such as integer ranges or directory hierarchies. Each role parameter has a data type, which is associated with a constraint domain. For example, in the role DC.access(pname=$?x$, data=$?y$), $?x$ has the data type corresponding to patient names and we can use equality constraints of the form $?x =$ 'Paul', and $?y$ has the data type corresponding to patient data categories, and we can use constraints of the form $?y \preceq \langle\text{medical}\rangle$.

Intuitively, a constraint domain is a domain of objects, such as numbers, points in the plane, or files in a file hierarchy, together with a language for speaking about these objects. The language is typically defined by a set of first-order constants, function symbols, and relation symbols. See [10] for a formal definition of constraint domains. For the purpose of this paper, it suffices to say that each constraint domain has a set of primitive constraints, and these primitive constraints can be conjuncted to form more complicated constraints. We now give several classes of constraint domains that have been defined in RT_1^C.

Tree domains Each constant of a tree domain takes the form $\langle a_1.a_2.\cdots.a_k\rangle$. Imagine a tree in which every edge is labeled with a string value. The constant $\langle a_1.\cdots.a_k\rangle$ represents the node for which $a_1.\cdots.a_k$ are the strings on the path from root to this node. A primitive constraint is of the form $x = y$ or $x\theta\langle a_1.\cdots.a_k\rangle$, in which x and y are variables and $\theta \in \{=, <, \leq, \prec, \preceq\}$.

The primitive constraint $x < \langle a_1.\cdots.a_k\rangle$ means that x is a child of the node $\langle a_1.\cdots.a_k\rangle$, and $x \leq \langle a_1.\cdots.a_k\rangle$ means that either $x = \langle a_1.\cdots.a_k\rangle$ or $x < \langle a_1.\cdots.a_k\rangle$. Similarly, the primitive constraint $x \prec \langle a_1.\cdots.a_k\rangle$ means that x is a descendant of $\langle a_1.\cdots.a_k\rangle$ (*i.e.*, the latter is a prefix of x), and $x \preceq \langle a_1.\cdots.a_k\rangle$ means that either $x = \langle a_1.\cdots.a_k\rangle$ or $x \prec \langle a_1.\cdots.a_k\rangle$.

Tree domains are used in the running example for the hierarchically organized data categories.

Discrete domains with sets Such a domain has a set of constants and one predicate $=$. A primitive constraint has the form $x = y$, or $x \in \{c_1, \ldots, c_\ell\}$, in which x and y are variables, and $c_1, \ldots, c_\ell$ are constants.

In our running example, the patient name is a discrete domain with sets. In our examples, we use the constraint $?x =$ 'Paul', which is a shorthand for $?x \in \{$'Paul'$\}$.

A constraint is a conjunction of primitive constraints, possibly from multiple constraint domains. Given a constraint $\phi(\boldsymbol{x})$, where $\boldsymbol{x}$ is a tuple of variables including all variables that occur free in ϕ, and a tuple $\boldsymbol{t}$ of constants, we say that $\phi(\boldsymbol{t})$ is satisfied if each primitive constraint in $\phi(\boldsymbol{x})$ evaluates to true when the variables in it are replaced with the corresponding constants in $\boldsymbol{t}$. For example, given the constraint $\phi(\langle x_1, x_2\rangle) = x_1 \leq \langle a_1.a_2\rangle \wedge x_2 \in (1, 10)$, and the tuple $\boldsymbol{t} = \langle\langle a_1.a_2.a_3\rangle, 2\rangle$, we have $\phi(\boldsymbol{t})$ is satisfied, because $\langle a_1.a_2.a_3\rangle \leq \langle a_1.a_2\rangle$ and $2 \in (1, 10)$ both evaluate to true.

Syntax In RT_1^C, a ground role is a role in which each parameter is constrained to be equal to one constant. A ground role defines a set of principals who are members of this ground role. Given a tuple of type-compatible constants $\boldsymbol{t}$, we use *members*$(A.r(\boldsymbol{t}))$ informally in the following to refer to the set of principals that are member of $A.r(\boldsymbol{t})$.

Credentials define role membership. (Here we use "credentials" to refer to both unsigned policy statements and to digitally signed credentials.) The variables that occur in a credential are local to that credential in the sense that they are implicitly universally quantified at the outermost level of the credential. In the following, $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{x}_1$, and $\boldsymbol{x}_2$ are tuples of variables that are all distinct. We now describe the four kinds of credentials in RT_1^C:

- *Type-1*: $A.r(\boldsymbol{x}) \longleftarrow B; \psi(\boldsymbol{x})$

  $A.r(\boldsymbol{x})$ is a role with each parameter being a variable, B is a principal, and $\psi(\boldsymbol{x})$ is a constraint over the variables in $\boldsymbol{x}$.
  This means that $B \in$ *members*$(A.r(\boldsymbol{t}))$, for any n-tuple of constants $\boldsymbol{t}$ such that $\psi(\boldsymbol{t})$ is satisfied.
- *Type-2*: $A.r(\boldsymbol{x}) \longleftarrow B.r_2(\boldsymbol{y}); \psi(\boldsymbol{x}, \boldsymbol{y})$

  $A.r(\boldsymbol{x})$ and $B.r_2(\boldsymbol{y})$ are both roles, and $\psi(\boldsymbol{x}, \boldsymbol{y})$ is a constraint (over the variables in $\boldsymbol{x}$ and $\boldsymbol{y}$).
  This means that

  $$\textit{members}(A.r(\boldsymbol{t})) \supseteq \textit{members}(B.r_2(\boldsymbol{s})),$$

  for every constant-tuple $\boldsymbol{t}$ and $\boldsymbol{s}$ such that $\psi(\boldsymbol{t}, \boldsymbol{s})$ is satisfied.
- *Type-3*: $A.r(\boldsymbol{x}) \longleftarrow A.r_1(\boldsymbol{y}).r_2(\boldsymbol{z}); \psi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$

  $A.r(\boldsymbol{x})$ and $B.r_1(\boldsymbol{y})$ are both roles, $r_2(\boldsymbol{z})$ is a role term, and $\psi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ is a constraint. We call $B.r_1(\boldsymbol{y}).r_2(\boldsymbol{z})$ a *linked role*.
  This means that

  $$\textit{members}(A.r(\boldsymbol{t})) \supseteq \textit{members}(D.r_2(\boldsymbol{w}))$$

  for all $D \in$ *members*$(B.r_1(\boldsymbol{s}))$ for every $\boldsymbol{t}$, $\boldsymbol{s}$, and $\boldsymbol{w}$ such that $\psi(\boldsymbol{t}, \boldsymbol{s}, \boldsymbol{w})$ is satisfied.
- *Type-4*: $A.r(\boldsymbol{x}_0) \longleftarrow A_1.r_1(\boldsymbol{x}_1) \cap A_2.r_2(\boldsymbol{x}_2);$
  $\psi(\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2)$

  $A.r(\boldsymbol{x}_0)$, $A_1.r_1(\boldsymbol{x}_1)$ and $A_2.r_2(\boldsymbol{x}_2)$ are roles, and $\psi(\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2)$ is a constraint. We call $A_1.r_1(\boldsymbol{x}_1) \cap A_2.r_2(\boldsymbol{x}_2)$ a *role intersection*.

This means that

$$\mathit{members}(A.r(\boldsymbol{s}_0)) \supseteq \ (\mathit{members}(A_1.r_1(\boldsymbol{s}_1)) \cap \mathit{members}(A_2.r_2(\boldsymbol{s}_2))),$$

for all constant-tuple $\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{s}_2$ such that $\psi(\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{s}_2)$ is satisfied.

We use σ to denote a credential, $Head(\sigma)$ to denote the role to the left of "$\longleftarrow$" in the credential σ, and $Body(\sigma)$ to denote the list of roles and constraints to the right of "$\longleftarrow$".

Semantics Given a set $\mathcal{P}$ credentials, its semantics is formally defined by translating each credential into a constraint datalog clause [9, 10, 14, 17]. We call the set of all resulting clauses the semantic program of $\mathcal{P}$.

Definition 1 (Semantic Program). Given a set $\mathcal{P}$ of policy statements, the *semantic program*, $SP(\mathcal{P})$, of $\mathcal{P}$, has one predicate r of arity $n+2$ for each n-ary role name r. Intuitively, $r(A, D, \boldsymbol{t})$ means that D is a member of the role $A.r(\boldsymbol{t})$. $SP(\mathcal{P})$ is the set of all constraint datalog clauses produced from policy statements in $\mathcal{P}$. The Semantic Program $SP(\mathcal{P})$ can be generated from $\mathcal{P}$ as follows.

For each $A.r(\boldsymbol{x})\longleftarrow D; \phi(\boldsymbol{x})$ in $\mathcal{P}$, add
$r(A, D, \boldsymbol{x}) :- \phi(\boldsymbol{x})$ $(m1)$
For each $A.r(\boldsymbol{x})\longleftarrow B.r_1(\boldsymbol{y}); \phi(\boldsymbol{x}, \boldsymbol{y})$ in $\mathcal{P}$, add
$r(A, z_1, \boldsymbol{x}) :- r_1(B, z_2, \boldsymbol{y}), z_1 = z_2, \phi(\boldsymbol{x}, \boldsymbol{y})$ $(m2)$
For each $A.r(\boldsymbol{x})\longleftarrow A.r_1(\boldsymbol{y}).r_2(\boldsymbol{z}); \phi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ in $\mathcal{P}$, add
$r(A, z_0', \boldsymbol{x}) :- r_1(A, y_1', \boldsymbol{y}),\ r_2(y_2', z_2', \boldsymbol{z}), y_1' = y_2', z_0' = z_2', \phi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ $(m3)$
For each $A.r(\boldsymbol{x})\longleftarrow B_1.r_1(\boldsymbol{y}) \cap B_2.r_2(\boldsymbol{z}); \phi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ in $\mathcal{P}$, add
$r(A, z_0', \boldsymbol{z}) :- r_1(B_1, z_1', \boldsymbol{y}),\ r_2(B_2, z_2', \boldsymbol{z}), z_0' = z_1', z_1' = z_2', \phi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ $(m4)$

An algorithm for evaluating a semantic program (which is a constraint Datalog program) is given in [10]. The algorithm requires using existential quantifier elimination to project constraints onto variables of interest. It is shown in [10] that existential quantifier elimination can be done efficiently in the three constraint domains mentioned in Section 4 and that the evaluation of constraint datalog programs such as $SP(\mathcal{P})$ is tractable when using these domains. However, the algorithm in [10] is a bottom-up algorithm that computes all logical implications of a semantic program. The algorithm is not goal-directed; thus, it is inefficient in practice and cannot deal with distributed storage of credentials.

5 Description of the Algorithms

Given a set of RT_1^C credentials, the goal of our algorithms is to answer the next three common kinds of queries:

1. Given a constrained role $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$, determine the set of principals that are members of the given constrained role and the associated constraints. More precisely, this query asks for a set of principal/constraint pairs Θ such that

(a) $\langle D, \varphi(\boldsymbol{x})\rangle \in \Theta$ implies $\varphi(\boldsymbol{x}) \Rightarrow \psi(\boldsymbol{x})$ and,
(b) for each principal D and each tuple of constants $\boldsymbol{t}$ such that $\psi(\boldsymbol{t})$ is satisfied, $SP(\mathcal{P}) \models r(A, D, \boldsymbol{t})$ if and only if there exists $\langle D, \varphi(\boldsymbol{x})\rangle \in \Theta$ such that $\varphi(\boldsymbol{t})$ is satisfied.

2. Given a principal D, determine a set of constrained roles that D is a member of. This query asks for a set Λ of constrained roles such that
(a) $A.r(\boldsymbol{x}); \varphi(\boldsymbol{x}) \in \Lambda$ implies $\varphi(\boldsymbol{x}) \Rightarrow \psi(\boldsymbol{x})$, and
(b) for each tuple of constants $\boldsymbol{t}$, $SP(\mathcal{P}) \models r(A, D, \boldsymbol{t})$ if and only if there exists $A.r(\boldsymbol{x}); \varphi(\boldsymbol{x}) \in \Lambda$ such that $\varphi(\boldsymbol{t})$ is satisfied.
3. Given a constrained role $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ and a principal D, determine the set of constraints under which D is a member of $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$. More precisely, this query asks for a set of constraints Ω such that
(a) $\varphi(\boldsymbol{x}) \in \Omega$ implies $\varphi(\boldsymbol{x}) \Rightarrow \psi(\boldsymbol{x})$, and
(b) for each tuple of constants $\boldsymbol{t}$ such that $\psi(\boldsymbol{t})$ is satisfied, $SP(\mathcal{P}) \models r(A, D, \boldsymbol{t})$ if and only if there exists $\varphi(\boldsymbol{x}) \in \Omega$ such that $\varphi(\boldsymbol{t})$ is satisfied.

To answer queries of the first form, we can apply a backward search algorithm starting from the constrained role $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$. For queries of the second form, we can apply a forward search algorithm starting from the principal D. For queries of the third form, we can use either a backward search or a forward search. For queries of the third form, we also have the alternative of using a bidirectional search algorithm, which simultaneously searches backwards from $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ and forwards from D.

When credential storage is distributed, the bidirectional search algorithm can find some chains that cannot be found by either forward or backward search alone.

5.1 The Backward Search Algorithm

The backward search algorithm constructs a proof graph, each node of which is given by (and represents) a constrained role $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$. (The nodes in this proof graph are called *role nodes*. This is the only kind of node constructed by the backwards search algorithm; the forward search algorithm also uses "principal nodes," which, as their name suggests, represent principals.)

Role nodes Each role node stores a set of solutions. A *solution* comprises a principal and a constraint. The algorithm maintains the following invariant. If the node $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ has a solution $\langle D, \varphi(\boldsymbol{x})\rangle$, then $\varphi(\boldsymbol{x}) \Rightarrow \psi(\boldsymbol{x})$ and $SP(\mathcal{P}) \models r(A, D, \boldsymbol{t})$ for each $\boldsymbol{t}$ such that $\varphi(\boldsymbol{t})$ is satisfied. When the algorithm terminates, it will also be the case that for each $\boldsymbol{t}$ such that $SP(\mathcal{P}) \models r(A, D, \boldsymbol{t})$, the node $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ has a solution $\langle D, \varphi(\boldsymbol{x})\rangle$ such that $\varphi(\boldsymbol{t})$ is satisfied. Thus, queries of type 1 can be answered by taking Θ to be the set of solutions associated with $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$. Similarly, queries of type 3 can be answered by taking Ω to be the set of constraints $\varphi(\boldsymbol{x})$ such that $\langle D, \varphi(\boldsymbol{x})\rangle$ is a solution associated with $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$. When there are certain kinds of edges between nodes, solutions can be propagated through the edges. Whenever a solution is about to be added to a node, we first check whether the solution is implied by a solution that already exists. A solution $\langle D, \varphi_1(\boldsymbol{x})\rangle$ is implied by a solution $\langle D, \varphi_2(\boldsymbol{x})\rangle$ if and only if $\varphi_1(\boldsymbol{x}) \Rightarrow \varphi_2(\boldsymbol{x})$. If so, then we do not add the new solution. Otherwise, the new solution is added, and the solution is propagated through outgoing edges.

The backward search algorithm maintains a queue of nodes that require further consideration, called the *backward processing queue*. The algorithm removes nodes from the queue one by one and processes them, repeating this until the queue is empty. Both the proof graph and the queue initially contain just one node, which corresponds to the query role. The algorithm also maintains a set of *backward expanded nodes*, which is initially empty.

To process a node $\eta_1 = A.r(\boldsymbol{x}); \psi_1(\boldsymbol{x})$, the algorithm does the following:

1. For each backward expanded node in the graph that has the form $\eta_2 = A.r(\boldsymbol{x}); \psi_2(\boldsymbol{x})$, it checks whether $\psi_1(\boldsymbol{x}) \Rightarrow \psi_2(\boldsymbol{x})$. If so, we know that all solutions for η_1 are also solutions to η_2, in which case we say that the node η_2 *subsumes* η_1. The algorithm adds a *specialization edge* from η_2 to η_1.
   The effect of this edge is that each solution $\langle D, \varphi_2(\boldsymbol{x})\rangle$ that is currently associated with or subsequently added to η_2 is propagated to η_1 as follows: let $\varphi_1(\boldsymbol{x}) = \psi_1(\boldsymbol{x}) \wedge \varphi_2(\boldsymbol{x})$; if $\varphi_1(\boldsymbol{x})$ is satisfiable, then add the solution $\langle D, \varphi_1(\boldsymbol{x})\rangle$ to η_1.
2. If no backward expanded node subsumes $\eta_1 = A.r(\boldsymbol{x}); \psi_1(\boldsymbol{x})$, then the algorithm adds η_1 to the set of backward expanded nodes, and examines all credentials defining $A.r(\boldsymbol{x})$. For each such credential, there are four cases.
   - *Case-1*: The credential takes the form
     $A.r(\boldsymbol{x}) \longleftarrow B; \psi_2(\boldsymbol{x})$

     Let $\varphi(\boldsymbol{x}) = \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x})$; if $\varphi(x)$ is satisfiable, then add the solution $\langle B, \varphi(\boldsymbol{x})\rangle$ to the node η_1.
   - *Case-2*: The credential takes the form
     $A.r(\boldsymbol{x}) \longleftarrow B.r_2(\boldsymbol{y}); \psi_2(\boldsymbol{x}, \boldsymbol{y})$

     Let $\psi_3(\boldsymbol{y}) = \exists \boldsymbol{x}\, [\, \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y})\,]$. If $\psi_3(\boldsymbol{y})$ is satisfiable (which can be determined by the existential quantifier elimination procedures of the constraint domains used in ψ_3), create a node $\eta_2 = B.r_2(\boldsymbol{y}); \psi_3(\boldsymbol{y})$, add it to the queue, and add an implication edge from η_2 to η_1.
     The effect of this edge is that each solution $[D, \varphi_2(\boldsymbol{y})]$ currently associated with or subsequently added to the node η_2 is propagated to η_1 as follows. Let $\varphi_1(\boldsymbol{x}) = \exists \boldsymbol{y}\, [\, \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y}) \wedge \varphi_2(\boldsymbol{y})\,]$. If $\varphi_1(\boldsymbol{x})$ is satisfiable, then add the solution $\langle D, \varphi_1(\boldsymbol{x})\rangle$ to the node η_1.
   - *Case-3*: The credential takes the form
     $A.r(\boldsymbol{x}) \longleftarrow A.r_1(\boldsymbol{y}).r_2(\boldsymbol{z}); \psi_2(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$

     Let $\psi_3(\boldsymbol{y}) = \exists \boldsymbol{x}\, \exists \boldsymbol{z}\, [\, \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})\,]$. If $\psi_3(\boldsymbol{y})$ is satisfiable, then create a node $\eta_2 = A.r_1(\boldsymbol{y}); \psi_3(\boldsymbol{y})$, add it to the queue, and create a backward monitoring edge from η_2 to η_1.
     The effect of the backward monitoring edge is that for each solution $\langle B, \varphi_1(\boldsymbol{y})\rangle$ currently associated with or subsequently added to the node η_2, the algorithm does the following. Let $\psi_4(\boldsymbol{z}) = \exists \boldsymbol{x}\, \exists \boldsymbol{y}\, [\, \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \wedge \varphi_1(\boldsymbol{y})\,]$. If $\psi_4(\boldsymbol{z})$ is satisfiable, create a node $\eta_3 = B.r_2(\boldsymbol{z}); \psi_4(\boldsymbol{z})$, add it to the queue, and create a linked implication edge from η_3 to η_1 with $\varphi_1(\boldsymbol{y})$ attached to it.

The linked implication edge from η_3 to η_1 does the following. Whenever a solution $\langle D, \varphi_3(\boldsymbol{z})\rangle$ is added to the node η_3, it is propagate to η_1 as follows. Let $\varphi_5(\boldsymbol{x}) = \exists \boldsymbol{y}\, \exists \boldsymbol{z}\, [\, \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \wedge \varphi_3(\boldsymbol{z}) \wedge \varphi_1(\boldsymbol{y})\,]$. If $\varphi_5(\boldsymbol{z})$ is satisfiable, then add the solution $\langle D, \varphi_5(\boldsymbol{x})\rangle$ to the node η_1.

– *Case-4*: The credential takes the form
$A.r(\boldsymbol{x}) \longleftarrow B_1.r_1(\boldsymbol{y}) \cap B_2.r_2(\boldsymbol{z}); \psi_2(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$

Let $\psi_3(\boldsymbol{y}) = \exists \boldsymbol{x}\, \exists \boldsymbol{z}\, [\, \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})\,]$, $\psi_4(\boldsymbol{z}) = \exists \boldsymbol{x}\, \exists \boldsymbol{y}\, [\, \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})\,]$. If both $\psi_3(\boldsymbol{y})$ and $\psi_4(\boldsymbol{z})$ are satisfiable, then create two nodes $\eta_2 = B_1.r_1(\boldsymbol{y}); \psi_3(\boldsymbol{y})$ and $\eta_3 = B_2.r_2(\boldsymbol{z}); \psi_4(\boldsymbol{z})$, add them to the queue, create an intersection edge from η_2 to η_1 with η_3 attached to it, and create an intersection edge from η_3 to η_1 with η_2 attached to it.

The effect of the intersection edge from η_2 to η_1 is that for each solution $\langle D, \varphi_1(\boldsymbol{y})\rangle$ currently associated with or subsequently added to the node η_2, the algorithm does the following. It examines the solutions of the node η_3. For each solution of η_3 taking the form $\langle D, \varphi_2(\boldsymbol{z})\rangle$, let $\varphi_5(\boldsymbol{x}) = \exists \boldsymbol{y}\, \exists \boldsymbol{z}\, [\, \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \wedge \varphi_1(\boldsymbol{y}) \wedge \varphi_2(\boldsymbol{z})\,]$. If $\varphi_5(\boldsymbol{x})$ is satisfiable, the algorithm adds the solution $\langle D, \varphi_5(\boldsymbol{x})\rangle$ to the node η_1.

5.2 The Forward Search Algorithm

The forward search algorithm constructs a proof graph that contains the following two kinds of nodes.

Principal nodes Each principal node corresponds to a principal; there is only one principal node for each principal.

Each principal node has a set of solutions. Each solution in a principal node is a constrained role. The invariant is that when the principal D has the solution $A.r(\boldsymbol{x}); \varphi(\boldsymbol{x})$, then we have $SP(\mathcal{P}) \models r(A, D, \boldsymbol{t}\,)$ for all $\boldsymbol{t}$ such that $\varphi(\boldsymbol{x})$ is satisfied.

Furthermore, when the algorithm terminates, it is also true that for each $\boldsymbol{t}$ such that $SP(\mathcal{P}) \models r(A, D, \boldsymbol{t}\,)$, the node D has a solution $\langle A.r(\boldsymbol{x});\ \varphi(\boldsymbol{x})\rangle$ such that $\varphi(\boldsymbol{t}\,)$ is satisfied. Thus, queries of type 2 can be answered by taking Λ to be the set of solutions associated with D. Similarly, queries of type 3 can be answered by taking Ω to be the set of constraints $\varphi(\boldsymbol{x})$ such that $\langle A.r(\boldsymbol{x});\ \varphi(\boldsymbol{x})\rangle$ is a solution associated with D.

Role nodes In the forward search algorithm, a role node is similar to that in the backward search algorithm. Each such node represents a constrained role $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ and contains a list of solutions of the form $\langle D, \varphi(\boldsymbol{x})\rangle$. The invariant here is as follows: if the node $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ has a solution $\langle D, \varphi(\boldsymbol{x})\rangle$, then $\varphi(\boldsymbol{x}) \Rightarrow \psi(\boldsymbol{x})$ and $SP(\mathcal{P}) \models r(A, D, \boldsymbol{t}\,)$ for each $\boldsymbol{t}$ such that $\varphi(\boldsymbol{t}\,)$ is satisfied.

Whenever a solution $\langle D, \varphi(\boldsymbol{x})\rangle$ is added to a role node $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$; it will find the principal node for D (and create one if one does not already exist), and add $A.r(\boldsymbol{x}); \varphi(\boldsymbol{x})$ as a solution to D. Notice that the invariant on the latter solution follows from the invariant on the former.

The forward search algorithm maintains a forward processing queue and proceeds by removing nodes from the queue and processing them one by one until the queue is empty. Initially, both the proof graph and the queue contain just a principal node. The forward search algorithm also maintains a set of forward expanded nodes, which is initially empty. Nodes are process as follows:

Forward processing a principal node D

1. Consider all *Type-1* credentials with the principal D in their bodies. For each such $A.r(\boldsymbol{x}) \longleftarrow D; \psi(\boldsymbol{x})$, the algorithm creates a role node $\eta_1 = A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$, adds η_1 to the forward processing queue, and adds the solution $\langle D, \psi(\boldsymbol{x})\rangle$ to η_1.
2. Each time a principal node D receives a new solution, the algorithm examines all credentials of *Type-4*. For each such $A.r(\boldsymbol{x}) \longleftarrow B_1.r_1(\boldsymbol{y}) \cap B_2.r_2(\boldsymbol{z}); \psi_1(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$, if the node D has two forward solutions taking the forms $\{B_1.r_1(\boldsymbol{y}); \varphi_1(\boldsymbol{y})\}$ and $\{B_2.r_2(\boldsymbol{z}); \varphi_2(\boldsymbol{z})\}$, respectively, and one of these is the new solution received by D, then the algorithm proceeds as follows. Let $\psi_2(\boldsymbol{x}) = \exists \boldsymbol{y}\ \exists \boldsymbol{z}\ [\, \psi_1(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \wedge \varphi_1(\boldsymbol{y}) \wedge \varphi_2(\boldsymbol{z})\,]$. If $\psi_2(\boldsymbol{x})$ is satisfiable, the algorithm creates the node $\eta = A.r(\boldsymbol{x}); \psi_2(\boldsymbol{x})$, adds η to the forward processing queue, and adds the solution $\langle D, \psi_2(\boldsymbol{x})\rangle$ to the node η.

Forward processing a role node $\eta_2 = B.r_2(\boldsymbol{y}); \psi(\boldsymbol{y})$**.**

In the following the effects of specialization, implication, and linked implication edges are to propagate solutions from one role node to another just as they do in the backward search algorithm.

1. If there exists a forward expanded node $\eta_1 = B.r_2(\boldsymbol{y}); \psi'(\boldsymbol{y})$ such that $\psi(\boldsymbol{y}) \Rightarrow \psi'(\boldsymbol{y})$, then add a specialization edge from η_2 to η_1. The node η_2 is removed from the queue.
2. If the node η_2 is still in the queue, add it to the set of forward expanded nodes, and examine all *Type-2* credentials with $B.r_2$ in the bodies. For each such credential $A.r(\boldsymbol{x}) \longleftarrow B.r_2(\boldsymbol{y}); \psi_2(\boldsymbol{x}, \boldsymbol{y})$, let $\psi_1(\boldsymbol{x}) = \exists \boldsymbol{y}\ [\, \psi(\boldsymbol{y}) \wedge \psi_2(\boldsymbol{x}, \boldsymbol{y})\,]$. If $\psi_1(\boldsymbol{x})$ is satisfiable, create a role node $\eta_1 = A.r_1(\boldsymbol{x}); \psi_1(\boldsymbol{x})$, add it to the forward processing queue, and add an implication edge from η_2 to η_1.
3. Create a principal node $\eta_1 = B$. Add a forward monitoring edge from η_2 to η_1.
   The effect of the forward monitoring edge is such that whenever the node η_1 receives a forward solution $A.r_1(\boldsymbol{x}); \varphi_1(\boldsymbol{x})$, the algorithm examines all credentials of *Type-3* with $A.r_1(\cdot).r_2(\cdot)$ in their bodies. For each such $A.r(\boldsymbol{z}) \longleftarrow A.r_1(\boldsymbol{x}).r_2(\boldsymbol{y}); \psi_1(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$, the algorithm proceeds as follows. Let $\psi_2(\boldsymbol{z}) = \exists \boldsymbol{x}\ \exists \boldsymbol{y}\ [\, \varphi_1(\boldsymbol{x}) \wedge \psi(\boldsymbol{y}) \wedge \psi_1(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})\,]$. If $\psi_2(\boldsymbol{z})$ is satisfiable, then create a node $\eta_3 = A.r(\boldsymbol{z}); \psi_2(\boldsymbol{z})$, add it to the forward processing queue, and add a linked implication edge from η_2 to η_3 with $\varphi_1(\boldsymbol{x})$ attached to it.

5.3 Bidirectional Search Algorithm

The bidirectional search algorithm integrates the backward and forward searches. The backward search algorithm and the forward search algorithm are executed simultaneously, starting with the query role and the query principal, respectively. As these two

searches progress, they typically construct some identical or related nodes. When this occurs, the bidirectional search transfers solutions between the backward proof graph and the forward proof graph. We transfer the solutions as follows:

- **Transfer solutions from the backward proof graph to the forward proof graph:** Whenever the role node $\eta_1 = A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ in the backward proof graph receives a solution, say $\langle D, \varphi(\boldsymbol{x})\rangle$, the algorithm creates the principal node $\eta_2 = D$ in the forward search graph, and adds the forward solution $A.r(\boldsymbol{x}); \varphi(\boldsymbol{x})$ to the node η_2.
- **Transfer solutions from the forward proof graph to the backward proof graph** For each pair of role nodes $\eta_1 = A.r(\boldsymbol{x}); \psi_1(\boldsymbol{x})$ in the forward graph and $\eta_2 = A.r(\boldsymbol{x}); \psi_2(\boldsymbol{x})$ in the backward graph, if $\psi_3(\boldsymbol{x}) = \psi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x})$ is satisfiable, then add a bidirectional monitoring edge from η_1 to η_2.
  The effect of the bidirectional monitoring edge is that whenever the node η_1 receives a solution $\langle D, \varphi_1(\boldsymbol{x})\rangle$, let $\varphi_2(\boldsymbol{x}) = \varphi_1(\boldsymbol{x}) \wedge \psi_2(\boldsymbol{x})$. If $\varphi_2(\boldsymbol{x})$ is satisfiable, then add the solution $\langle D, \varphi_2(\boldsymbol{x})\rangle$ to the node η_2.

Besides transferring the solutions between the backward proof graph and forward proof graph, we need to handle the role intersection specially.

- In the backward proof graph, if there is an intersection edge from η_2 to η_1, whenever the node η_2 receives a solution, say $\langle D, \varphi(\boldsymbol{x})\rangle$, the algorithm adds the principal node $\eta_3 = D$ to the forward processing queue.
- In the forward proof graph, whenever the principal node $\eta_1 = D$ receives a forward solution, say $B_1.r_1(\boldsymbol{y}); \varphi(\boldsymbol{y})$, the algorithm examines all credentials of *Type-4*. For each credential having either the form $A.r(\boldsymbol{x}) \longleftarrow B_1.r_1(\boldsymbol{y}) \cap B_2.r_2(\boldsymbol{z}); \psi_1(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ or the form $A.r(\boldsymbol{x}) \longleftarrow B_2.r_2(\boldsymbol{z}) \cap B_1.r_1(\boldsymbol{y}); \psi_1(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$, the algorithm proceeds as follows. Let $\psi_2(\boldsymbol{z}) = \exists \boldsymbol{x}\, \exists \boldsymbol{y}\, [\, \psi_1(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \wedge \varphi(\boldsymbol{y})]$. If $\psi_2(\boldsymbol{x})$ is satisfiable, the algorithm creates the role node $\eta_2 = B.r_2(\boldsymbol{z}); \psi_2(\boldsymbol{z})$ in the backward proof graph and adds η_2 to the backward processing queue.

6 Conclusions

RT_1^C is a language in the RT family of Role-based Trust-management languages. It features rich delegation structures, parameterized roles, and constraints. In this paper we present a goal-directed distributed credential chain discovery algorithm for RT_1^C. We describe this algorithm in detail and illustrate this algorithm with an example.

Comparing it with existing work on logic programming and deductive databases, our algorithm is a combination of the logic-programming style top-down query evaluation with tabling [3] (corresponding to our backward search) and the deductive database style bottom-up evaluation (corresponding to our forward search). Our algorithm uses hints provided through the storage types to determine which directions to use for a particular part of the proof; this enables the algorithm to touch only those credentials that are related to the query, which are likely to be a small fraction of all the credentials in the system.

Acknowledgement

Portions of this work were supported by NSF CCR-0325951, NSF CNS-0448204, NSF CCF-0524010, and sponsors of CERIAS. We thank the anonymous reviewers for their helpful comments.

References

1. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of CONCUR'97*, number 1256 in Lecture Notes in Computer Science, pages 135–150. Springer, 1997.
3. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, Jan. 1996.
4. D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.
5. C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. IETF RFC 2693, Sept. 1999.
6. C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software: Practice & Experience*, 30(15):1609–1640, Sept. 2000.
7. S. Jha and T. Reps. Analysis of SPKI/SDSI certificates using model checking. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 129–144. IEEE Computer Society Press, June 2002.
8. T. 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.
9. P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):26–52, Aug. 1995.
10. N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, number 2562 in LNCS, pages 58–73. Springer, Jan. 2003.
11. N. Li, J. C. Mitchell, and W. 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.
12. N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, Feb. 2003.
13. R. Ramakrishnan. Magic templates: a spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3-4):189–216, 1991.
14. P. Z. Revesz. Constraint databases: A survey. In L. Libkin and B. Thalheim, editors, *Semantics in Databases*, number 1358 in LNCS, pages 209–246. Springer, 1998.
15. R. L. Rivest and B. Lampson. SDSI — a simple distributed security infrastructure, Oct. 1996. Available at *http://theory.lcs.mit.edu/∼rivest/sdsi11.html*.
16. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.
17. D. Toman. Memoing evaluation for constraint extensions of Datalog. *Constraints: An International Journal*, 2:337–359, 1997.