Local Names In SPKI/SDSI 2.0

January 31, 2000

Abstract

In this paper, we analyze the notion of "local names" in SPKI/SDSI 2.0 and show that local names can be interpreted as distributed groups and distributed roles. Based on the distributed-group interpretation, we develop a simple logic program for SPKI/SDSI's linked local-name scheme and prove that it is equivalent to the name-resolution algorithm in SDSI 1.1 and the 4-tuple-reduction mechanism in SPKI/SDSI 2.0. This logic program is by itself a logic for understanding SDSI's linked local-name scheme. This logic has several advantages over previous logics, *e.g.*, those in [1] and [9]. For one thing, it is directly implementable. We have also enhanced our logic program to handle threshold functions and certificate reduction as well as certificate discovery.

We also discuss the use of local names for the purpose of authorization and show that they can be used in ways similar to roles in Role-Based Access Control (RBAC). We suggest several modifications to SPKI/SDSI 2.0 to make it simpler and cleaner. Among other things, we question the value of delegation certificates.

1 Introduction

Rivest and Lampson introduced "*linked local names*" in Simple Distributed Security Infrastructure (SDSI). This was motivated by the inadequacy of public-key infrastructures based on global name hierarchies, such as X.509 [4] and Privacy Enhanced Mail (PEM) [8]. After SDSI version 1.1 [11], the SDSI effort merged with the Simple Public Key Infrastructure (SPKI) effort. The result is SPKI/SDSI 2.0, about which the most up-to-date document is IETF RFC 2693 [7].

The goal of this paper is to study the notion of "local names." We give two interpretations for local names: distributed groups and distributed roles. Based on the distributed-group interpretation, we are able to give a simple logic program for SDSI's linked local-name scheme.

Existing work on logic for SDSI's linked local names includes the logic of Abadi [1] and Halpern and van der Meyden's Logic of Linked Name Containment (LLNC) [9]. Our logic program is by itself a logic for linked local names. Compared with existing work, our logic has the following features.

- It corresponds exactly to SPKI/SDSI 2.0. We prove the equivalence of our logic program, SDSI's name resolution algorithm, and SPKI's 4-tuple-reduction procedure. Our logic program also handles threshold functions, which are part of SPKI/SDSI 2.0.
- The logic program is runnable without modification. It can be executed using the XSB logic programming system [13], a Prolog variant. Furthermore, it is guaranteed to terminate for

a large class of queries, including, but not limited to, all the queries necessary for SPKI's evaluation of authorization requests.

- The logic program is quite simple. The version that doesn't handle threshold functions has only four rules and is significantly simpler than existing logics, *e.g.*, LLNC.
- The logic program can be easily enhanced to handle certificate discovery as well as certificate reduction. By certificate discovery, we mean the problem of finding relevant certificates among a potentially very large set of certificates and providing them in the order needed for SPKI's certificate reduction procedure. We give the enhanced program in this paper.

This logic program serves both as a logical definition and an implementation of certificate reduction with threshold functions. Our hope is that it will contribute the understanding of SPKI/SDSI's linked local-name scheme.

In this paper, we also discuss the way SPKI/SDSI uses local names for authorization purposes. The SPKI/SDSI work focuses more on data structures for certificate and infrastructure issues such as certificate distribution and revocation. Processing procedures of certificates are less clearly defined. The meaning and usage of different structures are not thoroughly discussed. In [7], the authors state: "The processing of certificates and related objects to yield an authorization result is the province of the developer of the application or system." Although we agree that data structures and infrastructures are very important, we disagree with this approach to certificate processing. We believe that the meaning and processing of certificates should be application-independent, rigorously defined, and extensively discussed so that people can understand the system; in this respect, we agree with the *trust-management approach* to authorization [3]. Only then can people who write policies have confidence that the policies have the intended meaning. Our discussion suggests several modifications to SPKI/SDSI 2.0 to make it simpler and cleaner. Among other things, we question the value of having delegation certificates.

The rest of this paper is organized as follows. In Section 2, we review SPKI/SDSI 2.0 and other related work on local names. In Section 3, we give the distributed-group interpretation of local names and our logic program for linked local-name resolution. We also compare our approach with existing work. In Section 4, we discuss the use of local names in authorization and show that local names can be used in the same way that "roles" are used in Role-Based Access Control (RBAC). We conclude in Section 5.

2 Background on Local Names

In this section, we give background information on "local names." In Section 2.1, we review SPKI/SDSI's linked local-name scheme. In this process, we occasionally refer to SDSI 1.1 [11] when there are differences between SPKI/SDSI 2.0 and SDSI 1.1. We review the logics of Abadi and Halpern and van der Meyden in Section 2.2 and Elien's work [6] on certificate discovery in Section 2.3.

2.1 Local Names in SPKI/SDSI 2.0

In SDSI, there are *principals* and *local names*. Principals are public keys and are therefore unique. Each principal has its own name space. A principal issues certificates to define local names in its name space. Besides principals and local names, SDSI 1.1 also has *special roots*. A special root is a name that is bound to the same principal in every name space. Although this notion seems to be meaningful, it is not present in SPKI/SDSI 2.0. For this reason, we do not deal with special roots in this paper, but it would not be difficult to add special roots to our logic program.

SDSI allows principals and local names to be linked together to form *compound names*. We call SDSI 1.1's compound names *general compound names*. A general compound name is a principal, a local name, an object of the form "(e)," or an object of the form "$e's\ \ f$," where e and f are general compound names.[1] The name-resolution algorithm in SDSI 1.1 always resolves compound names from left to right and ignores any parenthesis. Therefore, one can remove all parentheses from a compound name and still have an equivalent compound names. SPKI/SDSI 2.0 does not allow compound name to contain parentheses.

In most cases, a compound name starts with a principal. The only exception is when a compound name occurs inside a certificate, in which case it may start with a local name that is assumed to be in the certificate issuer's name space. One can always explicitly add the issuer to the front of the compound name. Therefore, every compound name can be transformed to a fully-qualified compound name defined as follows.

Definition 1 *A* **fully-qualified compound name** *has the form:*

$$keyA's\ \ name_1's\ \ name_2's\ \ \ldots\ \ name_p,$$

where $keyA$ *is a principal and each* $name_i$ *is either a local name or a principal.*

From now on, we use compound names to mean fully-qualified compound names unless we explicitly say the opposite.

SPKI/SDSI 2.0 has two kinds of certificates. *Name-definition certificates* came originally from SDSI; they bind a local name to a subject. *Delegation certificates* came originally from SPKI; they delegate some right from an issuer to a subject. Subjects can also contain threshold functions.

Definition 2 *A* **subject** *is either a compound name or an object having the form:*

$$\texttt{(k-of-n}\ K\ \ N\ \ sub_1\ \ sub_2\ \ \ldots\ \ sub_N\texttt{)}$$

where $K \leq N$ *are both positive integers and* sub_1, sub_2, $\ldots$, sub_N *are subjects.*

In most of this paper, we assume that subjects do not contain threshold functions, *i.e.*, that they are compound names. The handling of threshold functions will be discussed in Section 3.4. When it doesn't cause confusion, we use "certificates" to mean name-definition certificates. In this paper, these certificates are represented by the following form:

```
keyA says binds(name0, keyB's name1's ...  namep).
```

The principal `keyA` is the issuer of this certificate, and the compound name "`keyB's name1's ... namep`" is the subject. We say that this certificate *defines* the compound name "`keyA's name0`."

Compound names are eventually resolved to principals. SDSI 1.1 [11] gives a resolution algorithm, which we provide in Appendix C. Figure 1 gives an adapted version of it in C-style syntax. The adapted version removes the code for things that do not exist in SPKI/SDSI 2.0, namely group certificates, quote certificates, and encrypted objects, and it only deals with fully-qualified compound names. Otherwise, it is equivalent to the algorithm in SDSI 1.1, although we change the code to use only one function `REF` and to make the set of certificates an explicit argument, *i.e.*, the argument `Certs`.

[1]This is a simplified and syntactically sugared version of the actual syntax.

```
 1  REF(Certs, P’s n1’s n2’s ... nk) {
 2           if (k=0)  return P;
 3      else if (k=1) {
 4               if (n1 is a principal) return n1;
 5          else if (n1 is a local name and there exists a certificate
 6                    "binds(P’s n1, Q’s m1’s m2’s ... ml)" in Certs)
 7               return REF(Certs, Q’s m1’s m2’s ... ml)
 8          else fail;
 9      } else {
10          Q = REF(Certs, P’s n1)
11          return  REF(Certs, Q’s n2’s ... nk)
12      }
13  }
```

Figure 1: Adapted version of SDSI’s name-resolution algorithm

The algorithm in Figure 1 is nondeterministic. On lines 5 and 6, the algorithm is free to choose any certificate that defines the name “`P’s n1`.” Given a set of certificates $\mathcal{C}$ and a compound name cn, the function call “`REF(`$\mathcal{C}$`, `cn`)`” may return a principal, fail, or run forever. Given $\mathcal{C}$, the function `REF` defines a relation between compound names and principals.

Definition 3 *A compound name cn is* **resolvable** *to a principal k given a set of certificates* $\mathcal{C}$ *if and only if there exists an execution of* `REF(`$\mathcal{C}$`, `cn`)` *that returns k.*

SPKI/SDSI 2.0 defines a 4-tuple-reduction procedure to reduce compound names to principals. The following is taken from [7].

```
The rule for name reduction is to replace the name just defined by its
definition.  For example,
     (name K1 N N1 N2 N3) + [(name K1 N) -> K2]
          -> (name K2 N1 N2 N3)
or,
     (name K1 N Na Nb Nc) + [(name K1 N) -> (name K2 N1 N2 ... Nk)]
          -> (name K2 N1 N2 ... Nk Na Nb Nc)
```

The 4-tuple-reduction mechanism defines a relation among compound names. The goal of reduction is to reduce a compound name to a principal.

Definition 4 *A compound name* cn_1 *is* **reducible** *to a compound name* cn_2 *given a set of certificates* $\mathcal{C}$ *if and only if there exists a sequence of certificates in* $\mathcal{C}$ *that reduces* cn_1 *to* cn_2*.*

2.2 Overview of previous logics for local names

Both Abadi’s logic [1] and Halpern and van der Meyden’s Logic of Local Name Containment (LLNC) [9] aim at giving a logical account of linked local names. However, their goals are somewhat different.

Abadi’s goal is not to capture SDSI’s name-resolution algorithm exactly but rather to generalize it and to study axioms for linked local names in the generalized setting. He generalized the

Propositional Logic:	All instances of propositional tautologies
Reflexivity:	$p \longmapsto p$
Transitivity:	$(p \longmapsto q) \Rightarrow ((q \longmapsto r) \Rightarrow (p \longmapsto r))$
Left-monotonicity:	$(p \longmapsto q) \Rightarrow ((p's\ r) \longmapsto (q's\ r))$
Associativity:	$((p's\ q)'s\ r) \longmapsto (p's\ (q's\ r))$
	$(p's\ (q's\ r)) \longmapsto ((p's\ q)'s\ r))$
Key Globality:	$(k's\ g) \longmapsto g$
	if g is a global identifier, *i.e.*, a principal
Key Linking:	$(k\ \texttt{says}\ (n \longmapsto r)) \Rightarrow ((k's\ n) \longmapsto (k's\ r))$
	if n is a local name
Key Distinctness:	$\neg(k_1 \longmapsto k_2)$if k_1 and k_2 are distinct keys
Witnesses:	$\neg(p \longmapsto q) \Rightarrow \vee_k(\neg(p \longmapsto k) \wedge (q \longmapsto k))$
	$(p's\ q) \longmapsto k_1 \Rightarrow \vee_k((p \longmapsto k) \wedge (k's\ q \longmapsto k_1))$
Modus Ponens:	From ϕ and $\phi \Rightarrow \psi$ infer ψ

in all axioms, p, q, and r are compound names and k and k_1 are principals.

Figure 2: Axioms of LLNC

"resolvable" relation in SDSI, which is between compound names and principals, to a relation $\longmapsto$ among arbitrary compound principals. As shown by Halpern and van der Meyden [9], Abadi's logic draws conclusions about local names that do not follow from SDSI's name resolution algorithm. Abadi's generalizations lead to a quite complex axiom system, which include axioms of modal logics to reason about `says` and propositional logic tautologies for $\neg$ and $\wedge$. Studying axioms in this generalized setting may be an interesting problem by itself, but it is not needed to understand or to implement SDSI's linked local-name resolution.

Halpern and van der Meyden want to capture SDSI's name resolution exactly. However, they still use Abadi's generalized relation $\longmapsto$. The axioms for their logic LLNC are given in Figure 2. We think it is inadequate to use the generalized relation $\longmapsto$ for the purpose of capturing SDSI's name resolution. There is no query in SPKI/SDSI about the $\longmapsto$ relationship between two arbitrary compound names. Furthermore, using $\longmapsto$ leads to some problems that we now discuss.

The semantics for $\longmapsto$ is hard to define. The semantics for both logics maps compound names to sets of principals. The notation $[[p]]$ represents the set of principals that a compound name p maps to. The semantics for $\longmapsto$ is defined as the superset relation, which seems to be the only reasonable choice. So their semantics has the rule:

$$p \longmapsto q \quad \text{if and only if} \quad [[p]] \supseteq [[q]]$$

Abadi's axiomatization is not complete with respect to this semantics, but LLNC's is. To achieve this, LLNC has Witnesses Axioms, which are not in Abadi's logic.
Now consider the following example, in which one principal sees only the following three certificates:

```
keyAlice says binds(poker_buddies, keyTom).
keyAlice says binds(poker_buddies, keyJohn).
keyAlice says binds(classmates, keyJohn).
```

Then $[[keyAlice's\ poker\_buddies]] = \{keyTom, keyJohn\}$ and $[[keyAlice's\ classmates]] = \{keyJohn\}$. According to the above semantics, LLNC has to conclude:

$$keyAlice's\ poker\_buddies \longmapsto keyAlice's\ classmates$$

This conclusion can be derived in LLNC by using the first Witness axiom and propositional tautologies. We think that this conclusion is counter-intuitive. One intuitive reading of the relationship $p \longmapsto q$ is that p is somehow reducible (through 4-tuple reduction or some similar mechanism with more rules) to q. However, the name $keyAlice's\ poker\_buddies$ can not be reduced to $keyAlice's\ classmates$ given the above three certificates. Moreover, this conclusion is nonmonotonic. By nonmonotonic, we mean that if a principal sees only the above three certificates, it can derive this conclusion; but if it sees an additional fourth certificate, it may no longer do so. *E.g.*, consider adding the certificate "`keyAlice says binds(classmates, keyJack).`" We think that this nonmonotonicity is quite unacceptable in SDSI's distributed setting. All conclusions in SDSI are monotonic. It is often very difficult to know that one has all the certificates in a distributed environment. Therefore, one doesn't know whether the conclusion is valid from another principal's point of view or whether the conclusion will remain valid when the principal knows more information later.

Another disadvantage of using $\longmapsto$ is that, given one compound name p, there may exist an infinite number of compound names q's such that $p \longmapsto q$. For example, applying the Left-monotonicity axiom: $(p \longmapsto q) \Rightarrow ((p's\ r) \longmapsto (q's\ r))$
to $(alice's\ friends) \longmapsto (alice's\ friends's\ friends)$
results in an infinite sequence of new conclusions. This problem causes difficulty in implementing the logic. A query for all compound names q's such that $p \longmapsto q$ may not be answered in finite time. This is the nontermination problem described in [6], which we will review in Section 2.3.

Finally, these two logics do not handle threshold functions, which are part of SPKI/SDSI 2.0. Nor do they consider the certificate discovery problem.

2.3 Certificate reduction and discovery

SPKI/SDSI 2.0 defines the certificate-reduction procedure when certificates are provided in the right order. The requester needs to find the relevant certificates among a potentially very large set of certificates and to provide them in right order. In [6], Elien studied this certificate discovery problem.

Elien considered the certificate-discovery problem with name-definition certificates as well as delegation certificates and ACL entries. He considered the problem of finding a path of certificates that delegates a right R from a source principal to a destination principal given a set of certificates $\mathcal{C}$. Elien used the implication notation "$I \rightarrow S$" to represent all kinds of certificates. Each certificate is translated into an implication. In an implication, I and S are compound names; I is called the issuer, and S is called the subject. An implication "$K \rightarrow S$" means that the principal K delegates the right R to S. An implication "$(K's\ \ N) \rightarrow S$" means that the principal K defines its local name N to be S.

Certificate discovery can be done by inferring new implications from those that are translated from certificates. The general rule is as follows. Let I_1, I_2, S_1, S_2 be compound names.

Implication Chaining: From $(I_1 \rightarrow (I_2's\ \ S_1))$ and $(I_2 \rightarrow S_2)$, derive $(I_1 \rightarrow (S_2's\ \ S_1))$.

Elien showed that certificate discovery is a non-trivial problem. The following implication
$(K_1's\ \ A) \rightarrow (K_1's\ \ A's\ \ A)$
alone generates an infinite number of implications:

$(K_1's\ \ A) \rightarrow (K_1's\ \ A's\ \ A's\ \ A)$
$(K_1's\ \ A) \rightarrow (K_1's\ \ A's\ \ A's\ \ A's\ \ A)$
$\dots$

Nonetheless, he was able to provide a certificate-discovery algorithm that is guaranteed to terminate within polynomial time. His algorithm restricts the Implication Chaining rule to be used only when $|S_2| \leq |I_2|$. He proved that, with this restriction, all implications of the form "$K_0 \rightarrow K_1$" can still be generated. This restriction guarantees termination and polynomial complexity. Given a set R of n certificates, the total number of implications that can be derived from R is $O(n^3C)$, where C is the length of the longest compound name in R.

Actually, it is possible to go one step further. One can further restrict the Implication Chaining rule to be used only when $|S2| = 1$, *i.e.*, when S_2 is a principal. This more strict restriction can reduce the complexity to $O(n^2C)$ and still generate all implications of the form "$K_0 \rightarrow K_1$." To prove this, it is worth noting that the issuer of every implication is either a principal or a principal followed by a local name. With this observation, it is not difficult to modify Elien's proof to prove this result. We omit this proof, because it is not directly related to our main result.

3 A Logic Program for Linked Local Names

In this section, we present our logic program for SDSI's linked local-name scheme. We want to give a logic program that captures SPKI/SDSI's linked local-name schemes exactly, is as simple as possible, and can be used in practice. We think that these goals have been met. In this section, we first discuss the distributed-group interpretation of local names. In Section 3.2, we give a four-rule logic program to capture SDSI's linked local-name-resolution algorithm. In Section 3.3, we prove that our logic program is equivalent to the name-resolution algorithm of SDSI 1.1 and the 4-tuple-reduction mechanism of SPKI/SDSI 2.0. In Section 3.4, we enhance the program to handle threshold functions and certificate discovery in addition to name resolution. In Section 3.5, we compare this logic program with existing logics.

3.1 Local names as distributed groups

The algorithm `REF` in Figure 1 defines a "resolvable" relation between compound names and principals. Therefore, we can view compound names as groups of principals and the "resolvable" relation as a group-member relation. We use a binary predicate "*contains*" to represent this relation.

The information for determining the *contains* relation comes from name-definition certificates. We now assume that subjects of such certificates do not contain threshold functions; we will handle threshold functions in Section 3.4. A certificate "k `says binds(`n`,` cn`)`" means that any principal that the compound name cn contains is also contained by the compound name "$k's\ \ n$." Thus a name-definition certificate actually defines a superset-subset relationship, which we use the binary predicate "*includes*" to represent.

SDSI local names are really group names. Moreover, these groups are distributed. There isn't a central authority that manages all the groups. Each principal is in charge of defining its own groups. A principal does this by issuing name-definition certificates. An interesting point of SDSI is that one can use linked groups. A compound name is a linked group. The linked group "$keyA's\ \ name_1's\ \ name_2$" can be defined as:

$$keyA's\ \ name_1's\ \ name_2 = \bigcup \{\ KeyX's\ \ name_2 \mid KeyX \in keyA's\ \ name_1\ \}$$

Linking:	$contains([P0, N0 \mid T], P2) :- contains([P0, N0], P1), contains([P1 \mid T], P2).$
Superset:	$contains([P0, N0], P) :- includes([P0, N0], CN2), contains(CN2, P).$
Globality:	$contains([P0, P1], P1) :- isPrincipal(P1).$
Self-containing:	$contains([P0], P0) :- isPrincipal(P0).$

Figure 3: $\mathcal{P}_4$: Logic Program for Name Resolution [2]

3.2 A logic program for SDSI's name resolution

In our logic program, a fully-qualified compound name "$keyA's\ \ name'_1s\ \ name'_2s \ldots name_p$" is represented by a list: $[\ keyA,\ name_1,\ name_2,\ \ldots,\ name_p\ ]$. We assume that principals and local names are not encoded in lists and that principals can be distinguished from local names by a unary predicate "*isPrincipal*." A name-definition certificate is translated into a fact of the predicate "*includes*." For example:

```
keyA says binds (name0, keyB)
```
is translated to $includes([keyA,\ name_0],\ [keyB])$.
```
keyA says binds (name0, keyB's name1's name2)
```
is translated to $includes([keyA,\ name_0],\ [keyB,\ name_1,\ name_2])$.

The first version of our program $\mathcal{P}_4$ has four rules to infer about the predicate *contains* from facts of the predicate *includes* translated from certificates. These rules are shown in Figure 3.

The Linking rule implements the semantics of linked local names. The Superset rule enforces the semantics of *includes*. Actually, the relation *includes* is not strictly necessary. We can translate each certificate to a rule using only the predicate *contains*. For example:

```
keyA says binds (name0, keyB)
```
to $contains([keyA, name_0], keyB)$.
```
keyA says binds (name0, keyB's name1's name2)
```
to $contains([keyA, name_0], X) :- contains([keyB, name_1, name_2], X)$.

If we use this translation, the Superset rule won't be needed. However, we think it is clearer to have the predicate *includes*. It is also helpful in extending this program to handle certificate discovery. The Globality rule handles principals that occur inside a compound name; we will have more to say about this kind of compound names in Section 4. The only reason for the Self-containing rule is to handle certificates that have a principal as their subjects. They are translated into facts of the form "includes([keyA, name0], [keyB])." It is quite clear that the essence of SDSI's linked local-name scheme is the Linking rule.

Given a set of certificates $\mathcal{C}$, we can get a logic program $\mathcal{P}_{\mathcal{C}}$ as follows: Start with $\mathcal{P}_4$; add a fact of "*includes*" for each certificate in $\mathcal{C}$; finally add definitions for principals, by, for example, adding a fact "$isPrincipal(k)$" for each principal k that occurs in $\mathcal{C}$.

The program $\mathcal{P}_{\mathcal{C}}$ has a minimal Herbrand model, as defined in standard logic programming literature. The semantics of $\mathcal{C}$ is defined by this minimal model of $\mathcal{P}_{\mathcal{C}}$. For any atom "$contains(cn, k)$" in the minimal model of $\mathcal{P}_{\mathcal{C}}$, we can construct a proof sequence for the atom from $\mathcal{C}$.

Definition 5 *A* **proof sequence** *for an atom "$contains(cn, k)$" from a set of certificates $\mathcal{C}$ is a sequence of atoms: $a_1, a_2, \ldots, a_q$, where $a_q = contains(cn, k)$. Each atom a_i is the head of a ground*

[2] Note that we are using Prolog's syntax. Variables start an upper-case letter. The notation $[P0, N0|T]$ represents a list in which the first element is $P0$, the second element is $N0$, and the rest of the list is T.

instance R^{INST} of one of the four rules in $\mathcal{P}_4$, and each atom in the body of R^{INST} is either a fact in $\mathcal{P}_\mathcal{C}$ or appears as a_j, where $1 \leq j \leq i-1$.

Definition 6 *A compound name cn* **contains** *a principal k given a set of certificates $\mathcal{C}$ if and only if there exists a proof sequence for "$contains(cn, k)$" from $\mathcal{C}$.*

The minimal Herbrand model of $\mathcal{P}$ may be infinite, because we can construct an infinite number of compound names from just one principal and one local name. However, given any compound name *cn*, the set of principals k such that "$contains(cn, k)$" is true is finite, because the total number of principals is finite.

Readers familiar with logic programming might notice that the Linking rule is left-recursive and may never terminate in a backward-chaining inference engine, such as a Prolog engine. To deal with this problem, we use the XSB system [13], a logic programming system developed at SUNY Stony-Brook. XSB has several nice features that most Prolog systems do not have. One of them is tabling, which enables the handling of left-recursive programs.

Claim 1 *Given a program $\mathcal{P}_\mathcal{C}$, any query that consists of one atom of the predicate "contains" always terminates if the atom's first argument is a list with fixed number of elements.*

In [5], Chen and Warren proved that a query Q with a program $\mathcal{P}$ terminates under XSB's tabled evaluation if $\mathcal{P}$ has the bounded-term-size property. A stronger requirement is that there exists an upper bound on the size of the arguments of all goals generated while answering a query Q with a program $\mathcal{P}$. If this requirement is satisfied, only a finite number of goals will be generated; thus, the query terminates. If the query Q is a *contains* atom that has a list with a fixed number of elements as its first argument, let N_1 be the size of this argument and N_2 be the largest size of any compound names in $\mathcal{P}_\mathcal{C}$; then the size of any argument of any goal generated during answering Q is bounded by $O(\max(N_1, N_2))$.

We first give several examples of potentially nonterminating queries. A query of *contains* may not terminate if its first argument is a variable or a list that has a variable as its tail, *e.g.*, $[k0, n1|N]$. One such query is "$: -contains(CN, k1)$," where CN is a variable, and $k1$ is a principal. This query asks for a compound name that contains $k1$. It may not terminate, because there are an infinite number of compound names. Similarly, the query "$: -contains([k0|CN], k1)$" may not terminate either.

Note that the condition in Claim 1 is sufficient but not necessary. The query "$: -contains(alice, X)$" always fails, because the constant "*alice*" can not unify with any list. This query trivially terminates. The following are some terminating queries that are useful. Given a compound name *cn* and a principal k, the query "$: -contains(cn, k)$" determines whether *cn* contains k. Given a compound name *cn*, the query "$: -contains(cn, X)$" gives one principal that *cn* contains. To find all such principals, one can use the query "$: -findall(X, contains(cn, X), S)$," where the predicate $findall$ is a standard predicate in Prolog. Given principals $k0, k1$, the query "$: -findall(N, contains([k0, N], k1), N)$" gives the set of local names in $k0$'s name space that resolve to $k1$. This is useful when one wants to determine all the authorizations one principal gives to another. Such kinds of queries are very useful in writing, understanding, and debugging policies.

3.3 Equivalence results

The program $\mathcal{P}_4$ is a rather straightforward translation from the algorithm `REF` in Figure 1 into Prolog. Line 2 of the algorithm `REF` corresponds to the Self-containing rule. Line 4 corresponds to the Globality rule. Lines 5 to 7 correspond to the Superset rule. And Lines 10 and 11 correspond to the Linking rule. In the following, we formally state the equivalence of the algorithm `REF`, certificate reduction rules, and the logic program $\mathcal{P}_4$. The proofs are given in Appendix A.

Proposition 2 Equivalence of REF and $\mathcal{P}_4$: *A compound name cn is reducible to a principal k given a set of certificates $\mathcal{C}$ if and only if cn is resolvable to k given $\mathcal{C}$.*

Proposition 3 Equivalence of REF and *contains*: *A compound name cn is resolvable to a principal k given a set of certificates $\mathcal{C}$ if and only if the name cn contains the principal k.*

3.4 Handling threshold functions and certificate discovery

The name-resolution algorithm `REF` does not handle subjects with threshold functions. SPKI/SDSI 2.0 also does not give a clear definition of certificate reduction procedures involving threshold functions. We now extend the logic program in Figure 3 to handle them. A threshold function is represented by a term of the following form:

$$threshold(k, n, [\, sub_1,\ sub_2,\ \ldots\ , sub_n\,])$$

The program $\mathcal{P}_4$ in Figure 3 can do certificate reduction with name-definition certificates. To solve the certificate-discovery problem, we need to keep track of which certificates are used to derive a new conclusion. To do this, we add an extra argument to the predicates *contains* and *includes*. The third argument of the predicate *includes* is a certificate identifier. The third argument of the predicate *contains* is a list, which we call an evidence sequence. The evidence sequence for resolving a threshold-free subject *cn* to a principal *k* is a list of identifiers of those certificates that have been used in deriving the current conclusion. Our logic program returns the list in the same order as used in SPKI's certificate-reduction mechanism.

The evidence sequence for resolving a threshold function "`(k-of-n` $K\ \ N\ \ sub_1\ \ sub_2\ \ \ldots\ \ \ sub_N$`)`" to a principal $keyX$ is more complex than a list of certificate identifiers. We use the following form to represent it: $branch(K,\ [evidence(sub_{i_1}, seq_1),\ \ldots,\ evidence(sub_{i_K}, seq_K)]),$ where seq_j is the evidence sequence for resolving sub_{i_j} to $keyX$. It is straightforward to use evidence sequences to reduce subjects to principals.

The full XSB program that handles threshold functions and certificate discovery is given in Figure 4. This logic program provides a logical definition as well as an implementation for certificate reduction and discovery with threshold functions. In Appendix B, we give a logic program that encodes several name-resolution examples in Abadi [1] and Elien [6]. The two programs should be put together. Then one can load it into XSB and type queries.

Elien's certificate-discovery algorithm first generates all new implications and then checks whether the desired one is in it. This bottom-up evaluation mechanism is very inefficient when there are lots of certificates many of which are not relevant to the desired result.

Our logic programming approach puts the burden on the underlying logic programming system XSB. In this way, we can leverage extensive research in logic programming field. The XSB's table-based evaluation is like a query-oriented hybrid of top-down and bottom-up evaluation. It is more efficient than pure bottom-up evaluation, because unrelated conclusions are not generated.

```
:- table(contains/3).
:- import append/3 from basics.

contains([P0, N0 | T], P2, CertS) :-
    contains([P0, N0], P1, CertS1),
    contains([P1 | T], P2, CertS2),
    append(CertS1, CertS2, CertS).

contains([P0, N0], P, [Cert | CertS]) :-
    includes([P0, N0], CN2, Cert),
    contains(CN2, P, CertS).

contains([_P0, P1], P1, []) :- isPrincipal(P1).

contains([P], P, []) :- isPrincipal(P).

contains(threshold(_K, N, _SubjectList), _P, []) :-
    N < 0, !, fail.

contains(threshold(K, _N, _SubjectList), _P, [branch(0, [])]) :-
    0 >= K, !.

contains(threshold(K, N, [Subject | SubjectList]), P,
        [branch(K, [evidence(Subject,CertS1) | CertS2])]) :-
    contains(Subject, P, CertS1),
    K_1 is K - 1,
    N_1 is N - 1,
    contains(threshold(K_1, N_1, SubjectList), P, [branch(K_1, CertS2)]).

contains(threshold(K, N, [_Subject | SubjectList]), P, CertS) :-
    N_1 is N - 1,
    contains(threshold(K, N_1, SubjectList), P, CertS).
```

Figure 4: The XSB Program for Name Resolution

Elien's program can handle both name-definition certificates and delegation certificates. It is not difficult to add the code for delegation certificates. Because we are going to question the value of delegation certificates in SPKI/SDSI 2.0 in the next section, we stop with the current version of the program.

3.5 Comparison with LLNC

Now let us compare LLNC's axioms in Figure 2 with our rules in Figure 3. Our Self-containing rule is a limited version of LLNC's Reflexivity axiom. Our Globality rule is the same as LLNC's Key Globality axiom. Our Superset rule is a limited version of the Transitivity axiom. Our Linking rule is a limited version of the chaining of the Left-monotonicity axiom and the Transitivity axiom. The second Associativity axiom in LLNC is used implicitly when translating general compound names to fully-qualified compound names. The Key Linking axiom is used implicitly when translating name-definition certificates to facts of the predicate *includes*. The first Associativity axiom, the Key Distinctness axiom, the Witnesses Axioms, and propositional tautologies do not have counterparts in our logic program.

Our logic program can be viewed as a simplified version of LLNC's axioms, yet it is still enough to capture SPKI/SDSI's linked local-name scheme. This simplification is possible because of the use of two relations *contains* and *include* instead of one relation $\longmapsto$. The simplicity leads to direct implementation and is, we hope, easier to understand.

4 Discussions

In this section, we discuss the use of linked local names in authorization, show that local names can serve the function of roles in Role-Based Access Control (RBAC), and suggest several modifications of SPKI/SDSI 2.0.

Besides name-definition certificates, SPKI has delegation certificates and *access control lists (ACLs)*. An entry in an ACL is really a delegation from the issuer to the subject of the entry. Because ACL entries are always stored by their issuers and are never transmitted, they do not need to be signed and can only be used by their issuers. Otherwise, they are the same as delegation certificates.

4.1 Compound names in ACL entries

We've shown that compound names can be interpreted as distributed groups. Groups can be used to implement roles. Therefore, local names can be viewed as local roles.

In SPKI/SDSI 2.0, local names are used differently from roles. For example, the subject of an ACL entry can be any compound name or threshold function. In particular, the subject can be a principal. This is different from RBAC, in which authorizations can only be given to roles, and principals can only acquire authorizations through memberships in roles.

What if we impose the restriction that only local names be used as subjects of ACL entries? First, this restriction won't hurt expressive power. If an ACL entry has an arbitrary complex subject, one can always define a new local name and bind this new name to this complex subject. Moreover, this step makes sense. If a principal wants to grant some right to some complex subject, then the group of principals defined by this subject have some common meaning to this principal. It makes sense to define a local group for them.

After making this restriction, we can view local names as local roles. ACLs grant rights to local roles. Name-definition certificates define eligibility for local roles.

This change brings local names closer to existing paradigms such as groups and ACLs. This makes it easier for administrators to understand. We think this is very important. No matter how good a mechanism is, if it is very hard for the people who are going to write policies to understand it, it is going to be hard to deploy the mechanism widely.

4.2 Principals inside compound names

The syntax for SDSI's compound names allows principals to occur inside a compound name after a local name. For example, the following compound name is valid.

$$keyA's\ name_1's\ keyB's\ name_2$$

SDSI's resolution algorithm doesn't have a problem dealing with it. SPKI/SDSI's syntax also allows these kinds of compound names; its 4-tuple-reduction mechanism doesn't mention this case, but it would be trivial to modify the mechanism to handle this case.

What is the intuition for such a compound name? The principal $keyB$ resolves to itself in any name space. It doesn't matter what principals the name "$keyA's\ name_1$" resolves to; as long as it resolves to some principal, this whole compound name is equivalent to "$keyB's\ name_2$." But if "$keyA's\ name_1$" resolves to the empty set, then the name "$keyA's\ name_1's\ keyB's\ name_2$" also resolves to empty set. Therefore, the name definition

$$includes([keyC, name0], [keyA, name_1, keyB, name_2])$$

really means:

$$includes([keyC, name0], [keyB, name_2]) :- contains([keyA, name_1],\ X).$$

This seems to be the only reasonable reading of such a certificate. Although it might be useful in some scenarios, this is an obscure way to write a name definition. We believe that, if one really needs the ability to specify that a name binding is valid only when another compound name does not define an empty group, then, more likely, one needs other kinds of conditions for a name binding. What is needed is a mechanism to specify conditions on name definitions. Such a mechanism will yield clearer and more expressive policies. We recommend modifying the syntax of compound names to forbid principals to appear in the middle of compound names; we further recommend considering whether to support conditions for name definition.

4.3 Threshold functions in name definitions

We've given the logic program for handling threshold functions. But what is the intuitive interpretation of a threshold function? Consider the following example:

$$threshold(2, 3, [keyA's\ name_1,\ keyB's\ name_2,\ keyC's\ name_3]).$$

What group does this threshold function correspond to? According to the resolution algorithm, it is the set of principals who belong to at least two of the three groups: "$keyA's\ name_1$," "$keyB's\ name_2$," and "$keyC's\ name_3$." This is an unusual definition. In SPKI/SDSI 2.0, the most obvious way to use threshold functions is to simulate conjunctions and disjunctions. For example, "$threshold(k, k, cn_1, cn_2, \ldots, cn_k)$" is actually the intersection of all groups $cn_1, cn_2, \ldots, cn_k$, and $threshold(1, k, cn_1, ..., cn_k)$ is the union of all these groups. Because SPKI/SDSI 2.0 doesn't have the conjunction and disjunction operators, one has to implement them this way. We believe that it would be better to support conjunction and disjunction directly and therefore recommend adding these operators to SPKI/SDSI.

4.4 Are delegation certificates necessary?

The functionality of delegation certificates can be performed by ACLs and name-definition certificates. What value do delegation certificates add? Section 4.3 of RFC 2693[7] discusses this issue. The following example is given as a justification for delegation certificates.

Consider a firewall proxy for a network of DoD machines. The authors of [7] argue that using ACL on the firewall would require a gigantic ACL. But this is only true without name-definition certificates. The solution proposed in [7] uses delegation certificates. It uses an ACL to grant the access right to the key of the Secretary of Defense and also allows this key to further delegate. This can be done just as easily using name-definition certificates. Let `key_firewall` be the firewall proxy's public key; the firewall proxy can have an ACL containing an entry "`key_firewall's authorized_users`." Then `key_firewall` defines its group "`authorized_users`" to be "`key_Secretary_Defense's authorized_firewall_users`." The principal `key_Secretary_Defense` can in turn define its group "`authorized_firewall_users`" to include groups of other principals, and so on.

This example does not show that delegation certificates are necessary. Instead, we see that ACLs and local-name definitions can achieve delegation of authority. Actually, this is true in general. Delegating to a principal k without allowing it to further delegate can be achieved by putting k into a local group n that has the authority from an ACL. Furthermore, delegating to k and allowing it to further delegate can be achieved by including one of k's local groups in n. This can implement SPKI's boolean redelegation control.

Then, what additional values do delegation certificates offer? Our answer is "a different view of authorization." Delegation certificates give a per-right view. They allow the delegation of a specific right. When one principal `keyA` delegates some rights to another principal `keyB`, `keyB` has to delegate this right explicitly to other principals if the right is to propagate. For this to work, principals need to have a common understanding of authorizations.

Local names give a per-group view to authorization. Each member of a group has every right the group is entitled. In this sense, a group membership is a delegation of all authorities. One has to very careful when defining one local group to include a group of another principal.

Having two views of authorization can be helpful in some special cases. But, in general, it causes unnecessary confusion. It may be easier to achieve a common understanding of local names, because rights tend to relate to local resources, which may not be known by other principals.

We propose only giving authorization to local names. Principals acquire authorization through binding to local names. This use of local names is very similar to the notion of roles in RBAC. Existing RBAC normally assumes that there is a centrally defined role structure. However, local names are distributed and controlled by different principals. Therefore, they can be called distributed roles. This is very useful in scenarios where there is not a central authority, *e.g.*, e-commerce.

5 Conclusion

We have introduced a simple logic program for certificate reduction and discovery of SPKI/SDSI's linked local-name scheme and argued that it has some advantages over existing work. We have also discussed the use of "local names" in authorization and two interpretations for "local names," namely, distributed groups and distributed roles. We hope that this paper contributes to the understanding of local names, SPKI/SDSI, and trust management in general.

References

[1] M. Abadi, "On SDSI's Linked Local Name Spaces," Journal of Computer Security, 6(1-2), 1998, pp. 3–21.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A Calculus for Access Control in Distributed Systems," *ACM Transactions on Programming Languages and Systems*, 15(4), 1993, pp. 706–734.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The Role of Trust Management in Distributed Systems," in *Secure Internet Programming*, LNCS vol. 1603, Springer, Berlin, 1999, pp. 185-210.

[4] ITU-T Rec. X.509 (revised), *The Directory - Authentication Framework*, International Telecommunication Union, 1993.

[5] W. Chen and D. S. Warren, "Tabled Evaluation with Delaying for General Logic Programs," *Journal of the ACM*, 43 (1996), pp. 20–74.

[6] Jean-Emile Elien, "Certificate Discovery Using SPKI/SDSI 2.0 Certificates," Masters Thesis, MIT LCS, May 1998, `<http://theory.lcs.mit.edu/~cis/theses/elien-masters.ps>`.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI Certificate Theory," IETF RFC 2693, September 1999, `<ftp://ftp.isi.edu/in-notes/rfc2693.txt>`.

[8] S. T. Kent, "Internet Privacy Enhanced Mail," *Communications of the ACM*, 8 (1993), pp. 48–60.

[9] J. Halpern and R. van der Meyden, "A Logic for SDSI's Linked Local Name Spaces Preliminary Version," in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, 1999, pp. 111-122.

[10] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in Distributed Systems: Theory and Practice," *ACM Transactions on Computer Systems*, 10 (1992), pp. 265–310.

[11] R. Rivest and B. Lampson, "SDSI - A Simple Distributed Security Infrastructure," `<http://theory.lcs.mit.edu/~cis/sdsi/sdsi11.html>`.

[12] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-Based Access Control Models," *IEEE Computer*, 29(2), 1996, pp. 38-47.

[13] D. Warren, etc. "The XSB Programming System," `<http://www.cs.sunysb.edu/~sbprolog/xsb-page.html>`.

A Proofs

Name-definition certificates can be classified into two types.

Definition 7 *A type-1 name-definition certificate binds a local name in the issuer's name space to a principal. A type-2 name-definition certificate binds a local name to a compound name that is not a principal.*

Proposition 2 Equivalence of REF and certificate reduction: *A compound name cn is reducible to a principal k given a set of certificates $\mathcal{C}$ if and only if cn is resolvable to k given $\mathcal{C}$.*

Proof. By definitions, we need to prove that there exists a sequence of certificates $C_1, \ldots, C_p$ in $\mathcal{C}$ such that when $C_1, \ldots, C_p$ are applied to cn one after another, cn is reduced to k if and only if there exists an execution of `REF(`$\mathcal{C}$`,` cn`)` that returns k.

We first prove the only if part. Do induction on the length p of the sequence. Base case: p is zero, *i.e.*, cn is the principal k. Clearly, `REF(`$\mathcal{C}$`,` k`)` returns k. Induction step, case one, $cn = k_0's\ n_1$: Consider C_1. If C_1 is a type-1 certificate, then $p = 1$ and $C_1 =$`binds(`$k_0's\ n_1$`,` k`)`. Therefore, line 4 of `REF(`$\mathcal{C}$`,` cn`)` returns k. If C_1 is a type-2 certificate, then $C_1 =$`binds(`$k_0's\ n_1$`,` $k_1's\ m_1's\ \ldots\ m_r$`)`, and $C_2, \ldots, C_p$ reduce $k_1's\ m_1's\ \ldots\ m_q$ to k. By induction assumption, there is an execution of "`REF(`$\mathcal{C}$`,` $k_1's\ m_1\ \ldots\ m_r$`)`" that returns k. Therefore, line 5 to 7 of `REF(`$\mathcal{C}$`,` cn`)` may choose C_1 and returns k.

Induction step, case two, $cn = k_0's\ n_1's\ n_2's\ \ldots\ n_q$, where $q > 1$: Consider the sequence of compound names resulted from the reduction: $cn_0 = cn, cn_1, \ldots, cn_p = k$, where cn_i is the result of applying the certificate C_i to cn_{i-1}. Because each reduction step can only change the first two symbols in a compound name. The only way to shorten a compound name is by replacing two symbols with one principal. And compound names can only be modified from front. Then there exists an integer s such that $1 <= s < p$ and $cn_s = k_s's\ n2's\ \ldots\ n_p$. Therefore, the certificates $C_1, C_2, \ldots, C_s$ reduce $k_0's\ n_1$ to k_r and the certificates $C_{r+1}, \ldots, C_p$ reduce $k_r's\ n2's\ \ldots\ n_q$ to k. By induction assumption, corresponding executions of `REF` exist. Therefore, line 10 and 11 of `REF(`$\mathcal{C}$`,` cn may return k.

We now prove the if part. Suppose that there is an execution of `REF(`$\mathcal{C}$`,` cn`)` that returns k. Do induction on the number t of all recursive calls of `REF` in the execution. If t is one, then cn is either k or $k_1's\ k$. If t is greater than one, consider the first recursive call of `REF`. It happens either on line 7 or line 10. If it is on line 7, then cn has the form $k_0's\ n_1$. Let C_1 be the certificate chosen on lines 5 and 6 and cn_1 be the subject of C_1. Because the call on line 7 returns k with $t-1$ recursive calls, by induction assumption, there exist a sequence of certificates that reduces cn_1 to k. Therefore, C_1 followed by this sequence reduce cn to k. If the first recursive call happens on line 10, then cn has the form "$k_0's\ n_1's\ \ldots\ n_q$, where $q > 1$. Then the call `REF(`$\mathcal{C}$`,` $k_0's\ n_1$`)` (on line 10) returns a principal k_r within less than t recursive calls, and the call `REF(`$\mathcal{C}$`,` $k_r's\ n_2's\ \ldots\ n_q$`)` (on line 11) returns k within less than t calls. By induction assumption, there exist one sequence of certificates that reduce $k_0's\ n_1$ to k_r and another sequence of certificates that reduce $k_r's\ n_2's\ \ldots\ n_q$ to k. Concatenating two sequence together, they reduce cn to k. ∎

Proposition 3 Equivalence of REF and *contains*: *A compound name cn is resolvable to a principal k given a set of certificates $\mathcal{C}$ if and only if the name cn contains the principal k.*

Proof. First, let us prove the if part. If there is a sequence of proof steps that ends with $contains(cn, k)$. Do induction on the length of the sequence. Base case: when length is one, either Globality rule or Self-containing rule is used. In either case, cn is trivially resolvable to k. Consider the rule that is used in the last step. Again, cn is trivially resolvable to k if it is either the Self-containing rule or the Globality rule. If it is the Superset rule, then cn is of the form $k_0's\ \ n_1$, there is a certificate in $\mathcal{C}$ that is represented by "$includes([k_0, n_1], [k_1, m_1, ldots, m_q],$" and the atom $contains([k_1, m_1, \ldots, m_q], k)$ appears earlier in the proof sequence. By induction assumption, there is an execution of "`REF(`$\mathcal{C}$`,` $k_1's\ \ m_1's\ \ \ldots\ \ m_q$`)`" that returns k. Therefore, the call "`REF(`$\mathcal{C}$`,` $k_0's\ \ n_1$`)`" will go to line 5 through 7 and may return k. If the last step uses the Linking rule, then cn is of the form $k_0's\ \ n_1's\ \ n_2's\ \ \ldots\ \ n_q$; and $contains([k_0, n_1], k_1)$ and $contains([k_1, n_2, ldots, n_q], k)$ appear earlier in the sequence. By induction assumption, there exists an execution of "`REF(`$\mathcal{C}$`,` $k_0's\ \ n_1$`)`" that returns k_1 and there exists an execution of "`REF(`$\mathcal{C}$`,` $k_1's\ \ n_2's\ \ \ldots\ \ n_q$`)`" that returns k. Therefore, there exists an execution of "`REF(`$\mathcal{C}$`,` $k_0's\ \ n_1's\ \ n_2's\ \ \ldots\ \ n_q$`)`" that goes through lines 10 and 11 and returns k.

We now prove the only if part. Suppose that there is an execution of `REF(`$\mathcal{C}$`,` cn`)` that returns k. Do induction on the number t of all recursive calls of `REF` in the execution. If t is one, then cn is either k or $k_1's\ \ k$. The Self-containing rule and the Globality rule will prove $contains(cn, k)$. If t is greater than one, consider the first recursive call of `REF`. It happens either on line 7 or line 10. If it is on line 7, then cn has the form $k_0's\ \ n_1$. Let C_1 be the certificate chosen on lines 5 and 6 and cn_1 be the subject of C_1. Because the call on line 7 returns k with $t-1$ recursive calls, by induction assumption, there exist a proof sequence for $contains(cn_1, k)$. This sequence followed by $contains(cn, k)$ is a proof sequence for $contains(cn, k)$, the last step uses the Superset rule. If the first recursive call happens on line 10, then cn has the form "$k_0's\ \ n_1's\ \ \ldots\ \ n_q$, where $q > 1$. Then the call `REF(`$\mathcal{C}$`,` $k_0's\ \ n_1$`)` (on line 10) returns a principal k_r within less than t recursive calls, and the call `REF(`$\mathcal{C}$`,` $k_r's\ \ n_2's\ \ \ldots\ \ n_q$`)` (on line 11) returns k within less than t calls. By induction assumption, there exist one proof sequence for $contains([k_0, n_1], k_r)$ and one for $contains([k_r, n_2, ldots, n_q], k)$. Concatenating them together and add $contains(cn, k)$ to the end is a proof sequence for $contains(cn, k)$. The last step uses the Linking rule. ∎

B Examples

The following are several examples of name resolution. To use them, make one file that contains both these examples and the rules in Figure 4, then load the file into XSB and type in queries, *e.g.*, "query1(S)."

```
%%% Beginning of the example program

isPrincipal(prin(_X)).

% Example one, from Elien's master thesis.

% The following are from certificates in Section 1.5 of Elien's thesis.
includes([prin(k1), 'Grad_Student'], [prin(k1), 'Jean_Emile_Elien'], cert04).
includes([prin(k0), 'MIT'], [prin(k0), 'EECS', 'Student'], cert01).
includes([prin(k1), 'Student'], [prin(k1), 'Grad_Student'], cert03).
includes([prin(k0), 'EECS'], [prin(k1)], cert02).
includes([prin(k1), 'Jean_Emile_Elien'], [prin(k2)], cert05).

% The following is a certificate that uses recursive definition.
includes([prin(k0), 'MIT'], [prin(k0), 'MIT', 'MIT'], cert00).

% The query "query1(S)" returns a set of tuples (P,C) where P is a
% principal that "prin(k0)'s MIT" reduces to and C is the evidence sequence
% of the reduction, i.e., certificates used in the same order as in SPKI's
% 4-tuple reduction.  The following line is the result of the query.
% S = [(prin(k2)  ','  [cert01,cert02,cert03,cert04,cert05])]
% This also shows that the program can handle recursive name definitions.

query1(S) :- setof((P,C), contains([prin(k0), 'MIT'], P, C), S).


% Example two, from Abadi's paper.

includes([prin('K_self'), 'BrokersInc'], [prin('K1')], cert10).
includes([prin('K_self'), broker],
     [prin('K_self'), 'BrokersInc', 'NYoffice', 'Smith'], cert11).
includes([prin('K1'), 'NYoffice'], [prin('K2')], cert12).
includes([prin('K2'), 'Smith'], [prin('smith@aol.com')], cert13).

% Similar to "query1(S)," the query "query2(S)" asks for all principals
% and evidence sequence that "prin('K_self')'s broker" reduces to.
% The following line is the result of this query.
% S = [(prin(smith@aol.com)  ','  [cert11,cert10,cert12,cert13])]

query2(S) :- setof((P,C), contains([prin('K_self'), broker], P, C), S).


% Example three, to test threshold functions.

includes([prin(alice), trusted],
      threshold(2, 3, [[prin(alice), friends], [prin(alice), trusted, trusted],
                 [prin(alice), classmates]]), cert20).

includes([prin(alice), friends], [prin(bob)], cert21).
includes([prin(alice), friends], [prin(carl)], cert22).
includes([prin(alice), friends], [prin(david)], cert23).

includes([prin(alice), classmates], [prin(bob)], cert24).

includes([prin(bob), trusted], [prin(carl)], cert25).
includes([prin(david), trusted], [prin(david)], cert26).

% The query does similar things as in example two; however, the evidence
% sequence is more complex than a simple list.
% The following is the answer cut into several lines.  The two principals
% "prin(bob)" and "prin(carl)" are in "prin(alice)'s trusted".
% S = [(prin(bob) ',' [cert20,branch(2,[evidence([prin(alice),friends],[cert21]),
%                              evidence([prin(alice),classmates],[cert24])])]),
%      (prin(carl) ',' [cert20,branch(2,[evidence([prin(alice),friends],[cert22]),
%                                      evidence([prin(alice),trusted,trusted],
%                 [cert20,branch(2,[evidence([prin(alice),friends],[cert21]),
%                      evidence([prin(alice),classmates],[cert24])]),cert25])])])]

query3(S) :- setof((P,C), contains([prin(alice), trusted], P, C), S).

%%% Ending of the example program
```

C SDSI's name resolution algorithm

```
      ( ref: n1 n2 ... nk ) means REF(current principal,n1,n2,...,nk)
where
      REF(P,n1,n2,...,nk) =
          Q = P
          for i = 1 to k do Q = REF2(Q, (Local-Name: ni) )
          return Q
where:
      REF2(P,n) =
          if P is not of form ( Principal: ... ) ERROR
          if n = ( Principal: ... ) return n
          if n = ( Group: ... ) return n
          if n = ( Quote: y ) return n
          if n = ( Local-Name: y ) return REF2(P,lookup-value(P,y))
          if n = ( ref: n1 n2 ... nk ) return REF(P,n1,n2,...,nk)
          if n = ( Encrypt: ... ) return REF2(P,decrypt(n))
          if n = ( Assert-Hash: s h )
             then let t = REF2(P,s)
                  if hash(t) = h then return t else ERROR
          if n has the form name@a1.a2.....ak
             then return value of appropriate ( ref: ... ) form
                  according to special DNS name-handling rules
                  (This returns ERROR if P is not a local name-space.)
          else ERROR
where:
      lookup-value(P,y) = current value of y in P's name space.
```