Local Names In SPKI/SDSI*

Ninghui Li
Department of Computer Science
New York University
251 Mercer Street
New York, NY 10012, USA
ninghui@cs.nyu.edu
http://cs.nyu.edu/ninghui

Abstract

We analyze the notion of "local names" in SPKI/SDSI. By interpreting local names as distributed groups, we develop a simple logic program for SPKI/SDSI's linked local-name scheme and prove that it is equivalent to the name-resolution procedure in SDSI 1.1 and the 4-tuple-reduction mechanism in SPKI/SDSI 2.0. This logic program is itself a logic for understanding SDSI's linked local-name scheme and has several advantages over previous logics, e.g., those of Abadi [1] and Halpern and van der Meyden [13].

We then enhance our logic program to handle authorization certificates, threshold subjects, and certificate discovery. This enhanced program serves both as a logical characterization and an implementation of SPKI/SDSI 2.0's certificate reduction and discovery.

We discuss the way SPKI/SDSI uses threshold subjects and names for the purpose of authorization and show that, when used in a certain restricted way, local names can be interpreted as distributed roles.

1 Introduction

Rivest and Lampson introduced "*linked local names*" in the Simple Distributed Security Infrastructure (SDSI) [17]. They were motivated by the inadequacy of public-key infrastructures based on global name hierarchies, such as X.509 [11] and Privacy Enhanced Mail (PEM) [12]. After version 1.1, 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 documents are [7, 9, 10].

*This paper appears in Proceedings of the 13th IEEE Computer Security Foundations Workshop.

One goal of this paper is to study the notion of "local names." We give two interpretations for local names: distributed groups and distributed roles. Here, a group is a set of principals; a role is both a set of principals and a collection of permissions.

Based on the distributed-group interpretation, we are able to give a simple logic program for SDSI's linked local-name resolution. 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) [13]. Our logic program is itself a logic for linked local names. Compared with existing work, our logic has the following advantages.

- It corresponds exactly to SPKI/SDSI 2.0. We prove the equivalence of our logic program, SDSI's name resolution process, and SPKI's 4-tuple-reduction mechanism.
- The logic program can be run in any Prolog system with at most trivial modifications. We have run it using the XSB logic programming system (a Prolog variant) [19]. Under XSB, 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. It has only four rules and is significantly simpler than the two existing logics.

Moreover, the logic program can easily be enhanced to handle authorization certificates and threshold subjects. It can also easily be enhanced to handle certificate discovery in addition to 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 process.

We give the enhanced program in this paper. It serves both as a logical characterization and a functioning implementation of certificate reduction and discovery in SPKI/SDSI 2.0. Our hope is that it will contribute to the understanding of SPKI/SDSI's linked local-name scheme and authorization framework. We also believe that this logic program can be used in real applications, by using interfaces between XSB and other languages, *e.g.*, Java, C, Oracle, and ODBC (see [5] and XSB manual [19]).

The SPKI/SDSI work focuses more on data structures for certificates 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 [9], Ellison *et al.* 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 infrastructure issues 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 [4]. Only then can people who write policies have confidence that the policies have the intended meaning.

We discuss threshold subjects in SPKI/SDSI and propose adding an intersection operator in name certs. We also discuss the way SPKI/SDSI uses SDSI names for the purpose of authorization and describe a restricted version of SPKI/SDSI 2.0. This restricted framework is much simpler and "cleaner" than SPKI/SDSI 2.0 and still has significant expressive power. In this framework, local names can be interpreted as distributed roles.

The rest of this paper is organized as follows. In section 2, we review SPKI/SDSI 2.0 and other related work. In section 3, we give the distributed-group interpretation of local names and our logic program for linked local-name resolution. In section 4, we generalize the logic program to handle authorization certificates, threshold subjects, and certificate discovery. In section 5, we discuss the use of threshold subjects and SDSI names for the purpose of authorization. We conclude in section 6.

2 Background on Local Names

In this section, we give background information on "local names." In section 2.1, we review the linked local-name scheme in SPKI/SDSI 2.0 [7, 9, 10]. In this process, we occasionally refer to SDSI 1.1 [17] when there are differences between SPKI/SDSI 2.0 and SDSI 1.1. We review the logics of Abadi [1] and Halpern and van der Meyden [13] in section 2.2 and related work on certificate discovery [3, 7, 8] in section 2.3.

2.1 Linked local names in SPKI/SDSI

In SDSI, there are *principals* and *local identifiers*. Principals are public keys and are therefore unique. Each principal has its own name space. Besides principals and local identifiers, SDSI 1.1 also has *special roots*. A special root is an identifier 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 them to our logic program.

SDSI names are formed by linking principals and local identifiers. SDSI names as defined in SDSI 1.1 are more general than those in SPKI/SDSI 2.0, so we call them *general SDSI names*. A general SDSI name is a principal, a local identifier, an object of the form "(e)," or an object of the form "$e's\ \ f$," where e and f are general SDSI names.[1] The name-resolution procedure in SDSI 1.1 always resolves SDSI names from left to right and ignores any parenthesis. Therefore, one can remove all parentheses from a SDSI name and still have an equivalent name.

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

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

$$(a's\ \ m_1's\ \ m_2's \ldots m_q)$$

where a is a principal and each m_i is either a local identifier or a principal; the parentheses are optional.

From now on, we use *SDSI names* or *names* to mean fully-qualified SDSI names. In SPKI/SDSI 2.0, names must take the form of a principal followed by zero or more local identifiers, *i.e.*, the m_i's in the above definition can not be principals; we call these names *SPKI names*. We choose to use the this more general version of definition 1 in order to be compatible with SDSI 1.1.

We further define *local names* and *compound names*. A local name is a principal followed by a local identifier. A compound name is a SDSI name of length three or more.

SPKI/SDSI 2.0 has two kinds of certificates. *Name-definition certificates* (*name certs* for short) came originally from SDSI; a name cert binds a local name to the cert's subject. *Authorization certificates* (*auth certs*) came originally from SPKI; an auth cert delegates a certain permission from

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

```
1  REF(Certs, A's m1's m2's ... mq) {
2           if (q=0)  return A;
3      else if (q=1) {
4               if (m1 is a principal) return m1;
5          else if (m1 is a local name and there exists a name cert
6                   "A says binds(m1, B's n1's n2's ... nr)" in Certs)
7               return REF(Certs, B's n1's n2's ... nr);
8          else fail;
9      } else {
10         C = REF(Certs, A's m1);
11         return  REF(Certs, C's m2's ... mq);
12     }
13 }
```

Figure 1. Adapted version of SDSI's name-resolution procedure

a principal (the cert's issuer) to the cert's subject. There are also *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 auth certs. In this paper, we often use auth certs to mean both auth certs and ACL entries.

In SPKI/SDSI, the subject of a certificate can also be a k-of-n threshold subject.

Definition 2 *A* **subject** *is either a SDSI name or an object of the form:*

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

where K and N are both positive integers and $K \leq N$; each sub_i is a subject.

Different documents on SPKI/SDSI 2.0 differ on whether threshold subjects are allowed to appear in name certs. They are allowed in [9] and earlier versions of [10], but are not allowed in [7, 10]. In an earlier version of this paper, we deal with threshold subjects in name certs. In this paper we disallow them to conform to the newest SPKI documents. Instead, we deal with auth certs, which can have threshold subjects. There are more discussions of threshold subjects in section 5.1.

In this paper, name certs are represented in the following syntax:

$$a \text{ says binds } (m,\ b\text{'s}\ n_1\text{'s} \ldots n_r).$$

The principal a is the issuer of this name cert; m is the local identifier to be defined; and the name (b's n_1's ... n_r) is the subject. We say that this name cert *defines* the local name (a's m).

SDSI names are eventually resolved to principals. SDSI 1.1 [17] gives a resolution procedure. Figure 1 gives an adapted version of it in C-style syntax. This 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 SDSI names. Otherwise, it is equivalent to the procedure in SDSI 1.1, although we use one function `REF` instead of two functions `REF` and `REF2` and make the set of certificates an explicit argument, *i.e.*, the argument `Certs`.

The function `REF` in Figure 1 is nondeterministic. On lines 5 and 6, it is free to choose any name cert that defines the name (`A's m1`). Given a set of name certs $\mathcal{C}$ and a SDSI name sn, the function call `REF` $(\mathcal{C}, sn)$ may return a principal, fail, or run forever. Given $\mathcal{C}$, the function `REF` defines a binary relation between SDSI names and principals.

Definition 3 *A SDSI name "sn" is* **resolvable** *to a principal "b" given a set of name certs $\mathcal{C}$ if and only if there exists an execution of* `REF` $(\mathcal{C}, sn)$ *that returns "b."*

SPKI/SDSI 2.0 defines a 4-tuple-reduction mechanism to reduce SPKI names to principals. Recall that SPKI names have the form of a principal followed by a list of local identifiers. The following definition is from [9].

> The rule for name reduction is to replace the name just defined by its definition. For example,
>
> (name a m_1 ... m_q) + [(name a m_1) $\rightarrow b$]
> $\Longrightarrow$ (name b m_2 ... m_q)
>
> or,
>
> (name a m_1 ... m_q)
> + [(name a m_1) $\rightarrow$ (name b n_1 ... n_r)]
> $\Longrightarrow$ (name b n_1 ... n_r m_2 ... m_q)

The 4-tuple-reduction mechanism defines the following binary relation among SDSI names.

Definition 4 *A SPKI name "sn_1" is* **reducible** *to a SPKI name "sn_2" given a set of name certs $\mathcal{C}$ if and only if there exists a sequence of name certs in $\mathcal{C}$ such that when they are applied one after another to "sn_1" using the name-reduction rules, the final result is "sn_2."*

Note that although a reducible relationship is between two SPKI names, the goal of reduction is to reduce a SPKI

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:	$(a's\ b) \longmapsto b$
	if b is a global identifier, *i.e.*, a principal
Key Linking:	$(a\ \texttt{says}\ (m \longmapsto r)) \Rightarrow ((a's\ m) \longmapsto (a's\ r))$
	if m is a local name
Key Distinctness:	$\neg(a_1 \longmapsto a_2)$ if a_1 and a_2 are distinct keys, *i.e.*, principals
Witnesses:	$\neg(p \longmapsto q) \Rightarrow \vee_a(\neg(p \longmapsto a) \wedge (q \longmapsto a))$
	$(p's\ q) \longmapsto a_1 \Rightarrow \vee_a((p \longmapsto a) \wedge (a's\ q \longmapsto a_1))$
Modus Ponens:	From ϕ and $\phi \Rightarrow \psi$ infer ψ

in all axioms, p, q, and r are SDSI names and a and a_1 are principals.

Figure 2. Axioms of LLNC

name to a principal. Authorization process in SPKI/SDSI only needs to know the reducible relationships between SPKI names and principals.

2.2 Previous logics for linked local names

Both Abadi's logic [1] and Halpern and van der Meyden's Logic of Local Name Containment (LLNC) [13] aim at giving a logical account of SPKI/SDSI's linked local-name scheme. However, their goals are somewhat different.

Abadi's goal is not to capture SDSI's name-resolution procedure exactly but rather to generalize it and to study axioms for linked local names in his generalized setting. Abadi's logic deals with general SDSI names rather than fully-qualified SDSI names. Also, the main relation $\longmapsto$ in his logic is among general SDSI names, instead of between names and principals. These lead to a quite complex axiom system. Also, Abadi's logic draws conclusions about local names that do not follow from SDSI's name-resolution procedure, as shown by Abadi [1] and Halpern and van der Meyden [13]. Studying axioms in this generalized setting may be an interesting problem, but it is not needed to understand or to implement SDSI's linked local-name scheme.

Halpern and van der Meyden want to capture SDSI's name-resolution procedure 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 their purpose of capturing SDSI's name-resolution procedure. There is no query in SPKI/SDSI about the $\longmapsto$ relationship between two arbitrary SDSI 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 SDSI names to sets of principals. They use the notation $[[p]]$ to represent the set of principals that the name p maps to. In LLNC's semantics, $\longmapsto$ is defined as the superset relation, which seems to be the only reasonable choice. (For reasons I don't understand, Abadi [1] used a semantics that interprets $\longmapsto$ as the subset relation rather than the superset relation.) LLNC's semantics has the rule:

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

As claimed in [13], LLNC's axiomatization is complete with respect to this semantics. 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 name certs:

keyAlice says binds(friends, keyTom).
keyAlice says binds(friends, keyJohn).
keyAlice says binds(classmates, keyJohn).

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

$$keyAlice's\ friends \longmapsto keyAlice's\ classmates$$

This conclusion can be derived in LLNC by using the first Witness axiom and propositional tautologies. We believe 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 but more powerful mechanism) to q. However, it seems unlikely that the name "$keyAlice's\ friends$" can be reduced to "$keyAlice's\ classmates$" from 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 certificate, it may no longer do so. *E.g.*, consider adding the name cert "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 SDSI name p, there may exist an infinite number of SDSI names q's such that $p \longmapsto q$ is true. 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 SDSI name q's such that $p \longmapsto q$ is true may not be answered in finite time. This is the nontermination problem discussed by Elien [8]. In the following, we review Elien's work.

2.3 Certificate discovery

SPKI/SDSI 2.0 gives a certificate-reduction process 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 the right order. In [8], Elien, studied this certificate discovery problem and gave an algorithm for it. In a later paper [7], Elien *et al.*improved the algorithm.

Elien *et al.*considered the certificate-discovery problem with name certs as well as auth certs (including ACL entries). They considered the problem of finding a path of certificates that delegates a certain permission r from a source principal to a destination principal given a set of certificates. Each certificate, name or auth, is translated into an implication: $(I \longrightarrow S)$. In which I and S are SDSI names; I is called the issuer, and S is called the subject. The implication $(a \longrightarrow sn)$ means that the principal a delegates the permission r to sn; the implication $((a's\ m) \rightarrow sn)$ means that the principal a defines its local name $(a's\ m)$ to be sn.

Certificate discovery can be done by inferring new implications from those translated from certificates, in ways similar to 4-tuple reduction. Let I_1, I_2, S_1, S_2 be SDSI names.

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

They showed that certificate discovery is a non-trivial problem. The following implication

$$(a_1's\ m) \rightarrow (a_1's\ m's\ m)$$

alone generates an infinite number of implications:

$$(a_1's\ m) \rightarrow (a_1's\ m's\ m's\ m)$$
$$(a_1's\ m) \rightarrow (a_1's\ m's\ m's\ m's\ m)$$
$$\cdots\cdots$$

They solved this problem by utilizing the fact that ultimately one only needs implications of the form $(a_0 \rightarrow a_1)$, where a_0 and a_1 are principals. To generate all such implications, a restricted version of the Implication Chaining rule suffices. Their algorithm restricts the Implication Chaining rule to be used only when $|S_2| = 1$, *i.e.*, S_2 is a principal. They proved that all implications of the form $(a_0 \rightarrow a_1)$ can still be generated even with this restriction. This restriction guarantees termination and polynomial complexity. Given a set $\mathcal{C}$ of n certificates, the total number of implications that can be derived from $\mathcal{C}$ is $O(n^3L)$, where L is the length of the longest SDSI name in $\mathcal{C}$.

Their certificate-discovery algorithm in [7, 8] first generates all new implications and then checks whether the desired one is in it. This bottom-up evaluation mechanism can be very inefficient when there are lots of certificates many of which are not relevant to the desired result.

In [3], Aura also studied the certificate discovery problem. He used Delegation Networks (a special kind of graphs) to represent certificates and discussed efficient algorithms to search for a path in a Delegation Network. Aura argued that Delegation Networks typically have the hourglass shape with large number of servers and even larger number of clients connected through relatively small numbers of intermediate nodes and gave search algorithms optimized for Delegation Networks of this shape.

Aura's work [3] does not deal with linked local-name scheme in SPKI/SDSI; it only addresses certificate discovery in the "pre-SDSI version of SPKI." Delegation networks can represent auth certs that have a principal or a k-of-n-principal threshold as subject. But they can not represent auth certs that have SDSI names in subject, nor can they represent name certs. It is thus quite different from the work of Elien *et al.*and the goal of this paper.

3 A Logic Program for Linked Local Names

In this section, we present a four-rule logic program that captures exactly the linked local-name scheme of SPKI/SDSI 2.0. In this section, we first discuss the distributed-group interpretation of local names, then give the logic program to capture SDSI's linked local-name scheme. We also discuss the termination behaviour of this program and prove its equivalence to the name-resolution procedure of SDSI 1.1 and the 4-tuple-reduction mechanism of SPKI/SDSI 2.0.

Linking:	$contains([A0, M0, M1\|T], B)$:- $contains([A0, M0], A1)$, $contains([A1, M1\|T], B)$.
Superset:	$contains([A0, M0], B)$:- $includes([A0, M0], SN)$, $contains(SN, B)$.
Globality:	$contains([A0, B], B)$:- $isPrincipal(B)$.
Self-containing:	$contains([B], B)$:- $isPrincipal(B)$.

Figure 3. $\mathcal{P}_4$: A logic program for linked local-name resolution [2]

3.1 Local names as distributed groups

The function `REF` in Figure 1 defines a "resolvable" relation between SDSI names and principals. Since a name may be resolvable to multiple principals, it represents a group of principals. Thus, we can view the resolvable relation as a group-member relation. We use a binary predicate "$contains$" to represent this relation.

Let us now consider local names; recall that a local name is a principal followed by a local identifier. The groups defined by local names are distributed, in the sense that there isn't a central authority that manages all the groups. These groups are local to each principal. Each principal is in charge of defining its own groups; it does this by issuing name certs. An interesting point of SDSI is that one can use compound names to define local names. Compound names are "linked groups." For example, the compound name $(a's\ \ m_1's\ \ m_2)$ represents the following group:

$$\bigcup \{\ (x's\ \ m_2) \mid x \in (a's\ \ m_1)\ \}$$

The information for determining the $contains$ relation comes from name certs. A name cert "a says binds(m, sn)" means that any principal that the name "sn" $contains$ is also contained by the name $(a's\ \ m)$. Thus a name cert actually defines a superset relationship, which we use the binary predicate "$includes$" to represent.

3.2 A logic program for SDSI's name resolution

In our logic program, a fully-qualified SDSI name "$a's\ \ m_1's\ \ m_2's\ \ \ldots\ \ m_q$" is represented by a list: "$[\ a,\ m_1,\ m_2,\ \ldots,\ m_q\ ]$." We assume that principals and local identifiers are not encoded in lists and that principals are distinguishable from local identifiers by a unary predicate "$isPrincipal$." A name cert is translated into a fact of the predicate "$includes$." For example:

From "a_0 says binds(m, a_1)"
to "$includes([a_0,\ m],\ [a_1])$."
From "a_0 says binds (m, a_1's n_1's n_2)"
to "$includes([a_0,\ m],\ [a_1,\ n_1,\ n_2])$."

Our program $\mathcal{P}_4$ has four rules; they infer about the predicate $contains$ from facts of $includes$, which are translated from name certs. These rules are shown in Figure 3.

[2] Note that we are using Prolog's syntax. Variables start with an uppercase letter. The notation $[A0, M0|T]$ represents a list in which the first element is $A0$, the second element is $M0$, and the rest of the list is T.

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

From "a_0 says binds(m, a_1)"
to "$contains([a_0, m], a_1)$."
From "a_0 says binds (m, a_1's n_1's n_2)"
to "$contains([a_0, m],\ B)$:-
$contains([a_1, n_1, n_2],\ B)$."

If we use this translation, the Superset rule won't be needed. However, we think it is clearer to have the predicate $includes$. The distinction of $contains$ and $includes$ is also helpful when we extend the program $\mathcal{P}_4$ to handle certificate discovery. The Globality rule handles principals that occur inside a SDSI name. This rule can be removed if we only deal with SPKI names, *i.e.*, names of the form a principal followed by a list of local identifiers. We choose to have it in order to be compatible with SDSI 1.1. The Self-containing rule handles name certs that have a single principal as their subjects. They are translated into facts of the form $includes([a, m], [b])$. It is quite clear that the essence of SDSI's linked local-name scheme is the Linking rule.

Now let us compare the rules in $\mathcal{P}_4$ with LLNC's axioms (Figure 2). The Self-containing rule is a limited version of LLNC's Reflexivity axiom. The Globality rule is the same as LLNC's Key Globality axiom. The Superset rule is a limited version of the Transitivity axiom. The 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 SDSI names to fully-qualified SDSI names. The Key Linking axiom is used implicitly when translating name certs to facts of $includes$. Our logic program does not have counterparts for the first Associativity axiom, the Key Distinctness axiom, the Witnesses Axioms, and propositional tautologies in LLNC.

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

3.3 Termination results of query answering

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

The semantics of $\mathcal{P}_{\mathcal{C}}$ is defined by its 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 $contains(sn, b)$ in the minimal model of $\mathcal{P}_{\mathcal{C}}$, we can construct a proof sequence for it.

Definition 5 *A* **proof sequence** *for an atom $contains(sn, b)$ from a set of name certs $\mathcal{C}$ is a sequence of atoms: "$a_1, a_2, \ldots, a_q$," where $a_q = contains(sn, b)$. Each atom a_i is the head of a ground rule R^{INST} that is the ground instantiation 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 SDSI name "sn"* **contains** *a principal "b" given a set of name certs $\mathcal{C}$ if and only if there exists a proof sequence for the atom $contains(sn, b)$ from $\mathcal{C}$.*

The minimal Herbrand model of $\mathcal{P}_{\mathcal{C}}$ may be infinite, because we can construct an infinite number of SDSI names from just one principal and one local name. However, given any SDSI name sn, the set of principal b's such that $contains(sn, b)$ is true is finite, because the total number of principals is finite.

The program $\mathcal{P}_{\mathcal{C}}$ can be executed by any logic programming system to answer queries. If the query terminates, it will give the correct answer. However, readers familiar with Prolog might notice that the Linking rule is left-recursive; thus, a backward-chaining inference engine, such as a Prolog engine, may never terminate when using the program $\mathcal{P}_{\mathcal{C}}$ to answer queries. To deal with this problem, we use the XSB system [19], 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. Because of tabling, we can guarantee the termination of a large class of queries.

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 [6], 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 even stronger requirement is that there exists an upper bound on the size of the arguments of all goals generated during 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 q_1 be the size of this argument and q_2 be the size of the longest SDSI name in $\mathcal{P}_{\mathcal{C}}$; then the size of any argument of any goal generated during answering Q is bounded by $O(\max(q_1, q_2))$.

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. One such query is ":- $contains(SN, b)$," where SN is a variable, and b is a principal. This query asks for a SDSI name that contains b. It may not terminate, because there are an infinite number of SDSI names. Similarly, the query ":- $contains([a|SN], b)$" may not terminate either.

The following are some terminating queries. Let sn be a SDSI name and b be a principal. The query ":- $contains(sn, b)$" determines whether sn contains b. Each answer to the query ":- $contains(sn, Y)$" gives a principal that sn contains. To find all such principals, one can use the query ":- $findall(Y, contains(sn, Y), YS)$," where the predicate $findall$ is a standard predicate in Prolog; when this query returns, the variable YS will be instantiated to the list of all principals that are contained in sn. Given two principals a, b, the query

":- $findall([a, Z], contains([a, Z], b), ZS)$"

gives the set of all local names in a's name space that resolve to b. This can be useful when one wants to determine all the authorizations one principal gives to another. Such kinds of queries are useful in writing, understanding, and debugging policies.

3.4 Equivalence results

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

Proposition 2 Equivalence of 4-tuple reduction and REF: *A SPKI name "sn" is reducible to a principal "b" given a set of name certs $\mathcal{C}$ if and only if the name "sn" is resolvable to the principal "b" given $\mathcal{C}$.*

Proposition 3 Equivalence of REF and $\mathcal{P}_4$: *A SDSI name "sn" is resolvable to a principal "b" given a set of name* *certs $\mathcal{C}$ if and only if the name "sn" contains the principal "b" given $\mathcal{C}$.*

4 Handling certificate discovery, auth certs, and threshold subjects

In this section, we enhance the program $\mathcal{P}_4$ to handle certificate discovery, authorization certificates, and threshold subjects.

4.1 Handling certificate discovery

The program $\mathcal{P}_4$ in Figure 3 can do certificate reduction with name certs. Now we extend it to do certificate discovery as well, by keeping track of which name certs are used when deriving a new conclusion. To do this, we add a third argument to the two predicates $contains$ and $includes$. The third argument of the predicate $includes$ is a "certID"– a string constant that uniquely identifies the name cert from which this fact of $includes$ is translated. The third argument "es" of the atom $contains(sn, g, es)$ is a list of certID's. We call "es" an *evidence sequence* for $contains(sn, b)$; it is a list of certID's of those certificates that have been used in deriving $contains(sn, b)$. We then enhance the program $\mathcal{P}_4$ to generate evidence sequences. The enhanced program returns the evidence sequence in the same order as used in SPKI's certificate-reduction.

4.2 Handling auth certs

Now we further extend our logic program to handle auth certs and ACL entries in SPKI. An auth cert (including an ACL entry) is a delegation from its issuer to its subject about a certain permission. We use techniques from Delegation Logic (DL) [15, 16] to handle auth certs; an auth cert is represented using the following syntax:

$Issuer$ delegates $Right\hat{}D$ to Sub.

In which $Issuer$ is the principal that is the issuer of this cert. $Perm$ is a term that encodes the permission being delegated in this cert. Sub is the subject of this cert; in DL, it is called the delegatee of this delegation. D is $*$ if the subject is allowed to re-delegate this permission to others, and is 1 otherwise.

In our logic program, such an auth cert is represented as:

$authorizes(Issuer, Perm, D, Sub, CertID)$.

In which $CertID$ is a unique identifier for this auth cert.

To represent delegations of permissions that are derived through chaining certificates, we use another predicate $delegates$, which also takes give arguments :

$delegates(Sub, Perm, D, B, CertIDList)$.

In which Sub is a subject; it is a SPKI name. $Perm$ and D are the same as in the $authorizes$ predicate. B is the delegatee of this delegation. Note that B is required to be a principal; the $delegates$ predicate represents only delegations to a single principal. If one wants to know whether there is a delegation to a conjunction of multiple principals, *e.g.*, in the case of a co-signed request, one can use the dummy principal trick used in [16] to handle delegation queries that have a conjunction of principals as delegatee. $CertIDList$ is a list of CertID's that have been used to derive this delegation.

We could use one predicate instead of having both $authorizes$ and $delegates$. However, we choose to use two predicates to make things clearer, just as we choose to use both $includes$ and $contains$ for name resolution. Also note that the predicate $delegates$ in [16] has a different fifth argument from the predicate $delegates$ in this paper. In [16], $delegates$ does not have the CertIDList argument; there we were defining DL's semantics and were not concerned with certificate discovery. Instead, the predicate $delegates$ in [16] has a length argument. In this paper, we are able to get rid of the length argument because the boolean re-delegation control mechanism in SPKI/SDSI corresponds to delegation depth 1 and *, so we only need to distinguish between length 1 and *, which can be done by using the two predicates: $authorizes$ and $delegates$.

To allow reasoning of delegation relationships, we further add the following four rules.

1. $delegates(Sub, R, D, B, Cs)$:-
   $contains(Sub, A1, Cs1)$,
   $delegates(A1, R, D, B, Cs2)$,
   $append(Cs1, Cs2, Cs)$.

   This rule means that if a SDSI name Sub is resolvable to a principal $A1$ and $A1$ delegates to a principal B, then Sub delegates to the principal B. The effect of $contains(Sub, A1)$ is passing through every permission from Sub to $A1$. In fact, we interpret $contains(Sub, A1)$ as the speaks_for relationship "$A1$ speaks_for Sub" in DL [16]: .

2. $delegates(A, R, D, B, [C|Cs])$:-
   $authorizes(A, R, *, Sub1, C)$,
   $delegates(Sub1, R, D, B, Cs)$.

   This rule means that if a principal A delegates to a name $Sub1$ in an auth cert and allows it to re-delegate, and $Sub1$ delegates to a principal B, then the principal A also delegates to the principal B, whether B can re-delegate is determined by whether $Sub1$ allows B to do so.

3. $delegates(A, R, 1, B, [C|Cs])$:-
   $authorizes(A, R, 1, Sub1, C)$,
   $contains(Sub1, B, Cs)$.

This rule means that if a principal A delegates to $Sub1$ in an auth cert but doesn't allow it to re-delegate, and the name $Sub1$ is resolvable to a principal B, then the principal A delegates to the principal B, but doesn't allow B to re-delegate.

4. $delegates(B, \_R, \_D, B, []) : -isPrincipal(B)$.

   This rule means that every principal delegates to itself unconditionally.

The above rule 1, 2, and 3 are essentially instantiations of the delegation chaining rules in DL, except that we have added the evidence sequence for certificate discovery purpose. Rule 1, 2, and 3 are for the cases where the first delegation is a speaks_for relationship, a depth-$*$ delegation, or a depth-1 delegation, respectively.

Abadi interpreted the relation $\longmapsto$ in his logic [1] as a speaks_for relationship as well. He used the notion of speaks_for in [2, 14], which is similar to but not the same as that in [16]. His logic of linked local names has axioms of modal logics to enforce the meaning of speaks_for relationships, *i.e.*, $p \longmapsto q$ means that if q says something, then p says it too.

The speaks_for interpretation makes perfect sense when using SDSI names in authorization, in which case the speaks_for interpretation of $contains(sn, b)$ means that if the principal b makes a request, the request can be viewed as from sn. The effect is that b has all permissions that sn has.

However, we think this is not part of the name resolution, but rather an effect of interpreting SDSI names as groups of principals. In authorization, members of a group speak for the group. We believe that this speaks_for interpretation should be left out of name resolution.

Note that we use a term to represent the permission being delegated through an auth cert. Intersection of two permissions is computed automatically by unification. This should suffice for most applications. If one wants to compute intersection differently, it is always possible to provide a new predicate and change the rule 2 accordingly. For example, one can define a predicate $intersects$ and change the rule 2 to the following.

$delegates(A, R, D, B, [C|Cs])$:-
  $authorizes(A, R1, *, Sub1, C)$,
  $delegates(Sub1, R2, D, B, Cs)$.
  $intersects(R1, R2, R)$.

Note that our logic program does not deal with the validity of a certificate, this is the only field in SPKI's 5-tuples that is missing in our logic program. It is straightforward to enhance the program to handle validity period expressed by "not-before" and "not-after." However, SPKI/SDSI 2.0 also has online-test as a possible validity testing method, so we choose to leave validity testing out of the logic program.

4.3 Handling threshold subjects

We now extend our logic program to handle threshold subjects. A threshold subject is represented by a term of the following form:

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

The evidence sequence for a delegation from a threshold subject "$threshold(k, [\, sub_1,\ sub_2,\ \ldots\ , sub_n\,])$" to a principal b is more complex than a list of certID's. We use the following form to represent it:

$$branch(k,\ [\, [i_1, sub_{i_1}, es_1], \ldots, [i_k, sub_{i_k}, es_k]\,]),$$

where $i_j \in [1..n]$, sub_{i_j} is the i_j'th subject in the list of subjects in the threshold, and es_j is the evidence sequence for the delegation from sub_{i_j} to b. It is straightforward to use such an evidence sequence to reduce a threshold subject to a principal.

The full XSB program that handles name certs, auth certs, threshold subjects, and certificate discovery is given in Figure 4. We have added five rules to handle threshold subjects, so the program has 13 rules in total. This logic program provides a logical characterization as well as a functional implementation for certificate reduction and discovery with threshold subjects. This program together with some examples can be downloaded from the author's homepage: http://cs.nyu.edu/ninghui/software.

In section 2.3, we argued that the bottom-up approach in [7, 8] is 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. 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.

5 Discussions

In this section, we discuss the use of threshold subjects and SDSI names in authorization. We also show that when used in a restricted way, local names can be interpreted as distributed roles.

5.1 Threshold subjects in SPKI/SDSI

In [9] and earlier versions of [10], threshold subjects may appear in name certs. This practice is disallowed in [7] and the most up-to-date version of [10]. The authors of [7] explained this decision:

> The reason that a threshold subject may not appear in a name cert is that a name cert is used to define a name as a set of public keys; if a name

```
:- table(contains/3).
:- table(delegates/5).
:- import append/3 from basics.

contains([A0, M0, M1 | T], B, CertS) :-
    contains([A0, M0], A1, CertS1),
    contains([A1, M1 | T], B, CertS2),
    append(CertS1, CertS2, CertS).

contains([A, M], B, [Cert | CertS]) :-
    includes([A, M], SN, Cert),
    contains(SN, B, CertS).

contains([_A, B], B, []) :- isPrincipal(B).

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

delegates(threshold(K, SubList), R, D, B, CertS) :-
    !, delegates(threshold(K, 1, SubList), R, D, B, CertS).

delegates(threshold(K, _I, _SL), _R, _D, _B, [branch(0, [])]) :- K = 0, !.

delegates(threshold(K, _I, []), _R, _D, _B, []) :- K > 0, !, fail.

delegates(threshold(K, I, [Sub | SubList]), R, D, B,
        [branch(K, [[I, Sub, CertS1] | CertS2])]) :-
    delegates(Sub, R, D, B, CertS1),
    K_1 is K - 1,  I1 is I + 1,
    delegates(threshold(K_1, I1, SubList), R, D, B, [branch(K_1, CertS2)]).

delegates(threshold(K, I, [_Sub | SubList]), R, D, B, CertS) :-
    !, I1 is I + 1,
    delegates(threshold(K, I1, SubList), R, D, B, CertS).

delegates([A0|T], R, D, B, CertS) :-     % Delegates from a SDSI name
    contains([A0|T], A1, CertS1),
    delegates(A1, R, D, B, CertS2),
    append(CertS1, CertS2, CertS).

delegates(A, R, D, B, [Cert | CertS]) :- % Delegates from a principal
    authorizes(A, R, *, Sub1, Cert),
    delegates(Sub1, R, D, B, CertS).

delegates(A, R, 1, B, [Cert | CertS]) :- % Delegates from a principal
    authorizes(A, R, 1, Sub1, Cert),
    contains(Sub1, B, CertS).

delegates(B, _R, _D, B, []) :- isPrincipal(B).
```

Figure 4. The XSB Program for Name Resolution

cert could have a threshold subject as a subject the notion of the value of a name would become too convoluted to the usable in practice.

Now let us examine the intuitive meaning of a threshold subject in a name cert. Consider the following name cert:
"a_0 says binds(m_0, (k-of-n 2 3 (a_1's m_1) (a_2's m_2) (a_3's m_3)))."
It means that the group (a_0's m_0) should include the group defined by the threshold subject "(k-of-n 2 3 (a_1's m_1) (a_2's m_2) (a_3's m_3))." Then what group does this threshold subject define? It seems that the only reasonable interpretation is the set of principals who belong to at least two of the three groups: (a_1's m_1), (a_2's m_2), and (a's m_3). For example, the group "(k-of-n 2 3 (alice's friends) (bob's friends) (carl's friends))" means the set of principals who are the friends of at least two among alice, bob, and carl.

Threshold subjects can implement intersections. For example, the group "(k-of-n k k sn_1 sn_2 ... sn_k)" is the intersection of all groups sn_1, sn_2, ..., sn_k. In fact, there is no other way to express intersections in SPKI/SDSI 2.0. Without using threshold subjects in name certs, there is no way to define a local name to be the intersection of two other names. We think this is undesirably limited; SPKI/SDSI should support intersection in name certs through some mechanism, preferably an intersection operator. Note that an intersection operator can implement SPKI/SDSI's threshold subjects, because multiple name certs can express unions, and SPKI/SDSI threshold subjects are static in the sense that all subjects in a threshold subject are explicitly listed.

The meaning of threshold subjects in auth certs is different from that in name certs. In a name cert, k of the n subjects in a threshold subject must be resolved to a single principal. In an auth cert, k subjects can be resolved to different principals, as long as they further delegate to a single principal. For example, given the following certificates:

a_0 delegates $read(file1)$^* to (k-of-n 2 3 (a_1's m_1) (a_2's m_2) (a_3's m_3)).
a_1 says binds(m_1, a_4).
a_2 says binds(m_2, b).
a_4 delegates $read(file1)$^1 to b.

One can derive that "a_0 delegates $read(file1)$^1 to b"; whereas the group "(k-of-n 2 3 (a_1's m_1) (a_2's m_2) (a_3's m_3))" is empty because no principal belongs to more than one of (a_1's m_1), (a_2's m_2), and (a_3's m_3).

Therefore, the same threshold subject needs to be treated differently depending on whether it occurs in an auth cert or in a name cert. In our opinion, it is this difference that makes the notion of threshold subjects convoluted. Perhaps, this is also the reason why the designers of SPKI/SDSI 2.0 took threshold subjects out from name certs. An intersection operator in name certs provides similar functionality to (and perhaps is more intuitive than) threshold subjects without causing confusion, as it is syntactically different from threshold subjects.

5.2 Are auth certs necessary?

The functionality of authorization certificates can be performed by ACLs and name certs. What value do auth certs add? This issue is discussed in section 4.3 of RFC 2693 [9], and the following example is given as a justification for auth certs.

Consider a firewall proxy for a network of DoD machines. The authors of [9] argue that using ACL on the firewall would require a gigantic ACL. But this is only true without using name certs. The solution proposed in [9] uses auth certs. It uses an ACL to grant the access permission to the key of the Secretary of Defense and also allows this key to further delegate. This can be done as easily without using auth certs. Let keyFW be the firewall proxy's public key; and let keySD be the public key of the Secretary of Defense. Then the firewall proxy creates an ACL entry authorizing "keyFW's authdUsers" and issues a name cert: "keyFW says binds(authdUsers, keySD's authdFWUsers)." The principal keySD can in turn define the group "keySD's authdFWUsers" to include groups of other principals, and so on.

This example does not show that auth certs are necessary. Instead, we see that ACLs and name certs can achieve delegation of authority. Actually, this is true in general. Delegating to a principal b without allowing it to further delegate can be achieved by putting b into a local group m that has the authority from an ACL or from being member of another principal's local group. Furthermore, delegating to b and allowing it to further delegate can be achieved by including one of b's local groups in m. This can implement SPKI's boolean re-delegation control. Note that we do not even need compound names to achieve this, just local names suffice.

Auth certs may contain threshold subjects. Without auth certs, threshold subjects can only be used in the roots of authority, *i.e.*, ACL entries. A principal can not delegate a permission it gets from others to a threshold subject. Besides the ability to delegate to threshold subjects, what additional values do auth certs offer? Our answer is "a different view of authorization." Auth certs give a per-permission view. They allow the delegation of a specific permission. When principal a delegates a certain permission to principal b, b has to delegate this permission explicitly to other principals if the permission is to propagate. For this to work, principals need to have a common understanding of permissions.

Local names give a per-group view to authorization. Each member of a group has every permission the group is entitled. In this sense, a group membership is a delegation of all authorities. Thus, one has to be very careful when defining one local group to include a group of another principal. In this framework, principals need to have a common understanding of local names.

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 permissions tend to be related to local resources, which may not be known by other principals. Moreover, SPKI/SDSI already uses linked local names, which only makes sense when there is some common understanding on local names.

5.3 Local names as distributed roles

We have discussed the interpretation of local names as distributed groups. Groups can be used to implement roles. Can we interpret local names as roles?

In Role-Based Access Control [18], a role is both a collection of entities and a collection of permissions. The role serves as an intermediate layer between entities and authorizations. Entities in a system can only get permissions by adopting certain roles.

In SPKI/SDSI 2.0, the subject of an ACL entry can be a local name, a principal, or a threshold subject. Thus, permissions can be given to principals directly. This is different from roles in RBAC, in which permissions can only be given to roles.

Now let us create an authorization framework by restricting SPKI/SDSI. We make three restrictions: no auth certs are allowed; the subject of an ACL entry must be a local name; and the subject of a name cert must be a principal or a local name. With these restrictions, we get a subset of SDSI 1.1. Although this framework seems very limited, it has quite some expressive power. As we have discussed before, it can still implement delegation of authorities and boolean re-delegation control.

Another advantage of this framework is that local names can be viewed as roles. In this framework, there are only three kinds of credentials. An ACL entry grants a certain permission to a role of the form (a's m). A name cert "a says $\mathrm{binds}(m,\ b)$" means that the principal b is a member of the role (a's m). A name cert "a says $\mathrm{binds}(m,\ b$'s $n)$" defines the role dominance relationship: the role (b's n) dominates the role (a's m), *i.e.*, the role (b's n) has all the permissions that the role (a's m) has, or equivalently, the role (a's m) contains all the principals in the role (b's n). This framework is quite simple and is hopefully easy to understand.

Note that local names are different from roles in traditional RBAC in some aspects. In traditional RBAC, control of role membership and role permissions is typically centralized to a few users and there is a centrally defined role hierarchy. Local names are distributed and controlled by different principals. In this sense, they are distributed roles. This is useful in scenarios where there is no central authority, *e.g.*, e-commerce.

RBAC also has the notion of sessions. An entity can create a session and choose to activate some subsets of the entity's roles in one session. In this restricted SDSI framework, since principals are responsible for providing certificates, they can choose to provide only enough certificates to prove their membership in a subset of all its roles.

There seems to be no easy way to implement RBAC's constraints in this framework. This is partly because of the distributed nature of local names and the non-monotonic nature of constraints.

In this framework, local names are brought closer to existing paradigms such as groups and roles. 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 is going to be hard to deploy the mechanism widely.

6 Conclusion

We gave a simple logic program to capture SPKI/SDSI's linked local-name scheme. We also gave an enhanced version of the program; it serves both as a logical characterization and as an implementation of SPKI/SDSI's certificate reduction and discovery. We discussed the use of threshold subjects and names in authorization and proposed a restricted way to use SPKI/SDSI in which local names can be interpreted as distributed roles. We hope that this paper contributes to the understanding of local names, SPKI/SDSI, and trust management and distributed authorizations in general.

Acknowledgement

Joan Feigenbaum and Martin Abadi made many useful comments on early drafts of this paper. Email discussions with Carl Ellison were helpful. I'd also like to thank anonymous reviewers for their helpful reports.

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 (1993), pp. 706–734.

[3] T. Aura, “Fast Access Control Decisions from Delegation Certificate Databases,” in *Proceedings of the 3rd Austrilian Conference on Information Security and Privacy ACISP’98*, LNCS vol. 1438, Springer, Berlin, 1998, pp. 284-295.

[4] 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.

[5] M. Calejo, “InterProlog: A Java front-end and enhancement for Prolog,” http://dev.servisoft.pt/interprolog/.

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

[7] D. Clarke, J. Elien, C. Ellison, Fredette, A. Morcos, and R. Rivest, “Certificate Chain Discovery in SPKI/SDSI,” Draft Paper, Nov 1999, http://theory.lcs.mit.edu/~rivest/publications.html.

[8] J. 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.

[9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, “SPKI certificate theory,” IETF RFC 2693, September 1999.

[10] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, “Simple Public Key Certificate,” Internet Draft (Work in Progress), July 1999, http://world.std.com/~cme/spki.txt.

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

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

[13] 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*, IEEE Computer Society Press, Los Alamitos, 1999, pp. 111-122.

[14] 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.

[15] N. Li, J. Feigenbaum, and B. Grosof, “A Logic-Based Knowledge Representation for Authorization with Delegation (Extended Abstract),” in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, Los Alamitos, 1999, pp. 162-174. Full paper available as IBM Research Report RC21492.

[16] N. Li, B. Grosof, and J. Feigenbaum, “A Practically Implementable and Tractable Delegation Logic,” to appear in *Proceedings of the 21st IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos, 2000.

[17] R. Rivest and B. Lampson, “SDSI - A Simple Distributed Security Infrastructure,” October 1996, http://theory.lcs.mit.edu/~cis/sdsi/sdsi11.html.

[18] R. Sandhu, “Role-Based Access Control Models,” *Advances in Computers*, vol. 46, Academic Press, 1998. http://www.list.gmu.edu/articles/advcom/a98rbac.ps.

[19] D. Warren, etc. “The XSB Programming System,” http://www.cs.sunysb.edu/~sbprolog/xsb-page.html.

A Proofs

To prove propositions 2 and 3, we classify name certs into two types.

Definition 7 *A* type-1 name cert *binds a local name in the issuer’s name space to a principal. A* type-2 name cert *binds a local name to a SDSI name other than a principal.*

Proposition 2 Equivalence of 4-tuple reduction and REF: *A SPKI name “sn” is reducible to a principal “b” given a set of name certs $\mathcal{C}$ if and only if the name “sn” is resolvable to the principal “b” given $\mathcal{C}$.*

Proof. 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 sn one after another, sn is reduced to b if and only if there exists an execution of `REF(`$\mathcal{C}, sn$`)` that returns b (see Figure 1 for the code of `REF`).

We first prove the only if part by induction on the length p of the cert sequence. Base step: p is zero, then sn is the principal b. Clearly, `REF(`$\mathcal{C}, b$`)` returns b.

Now the induction step. When sn is a local name, let sn be $(a_0's\ \ m_1)$; consider the name cert c_1. If c_1 is a type-1 name cert, then p must be 1, and c_1 must be “a_0 says $\mathrm{binds}(m_1,\ b)$.” Therefore, there is an execution of `REF(`$\mathcal{C}, sn$`)` that chooses c_1 on lines 5-6 and returns b on line 7. If c_1 is a type-2 name cert, let c_1 be “a_0 says $\mathrm{binds}(m_1,\ sn_1)$,” then the certificate sequence “$c_2, \ldots, c_p$” reduce “sn_1” to b. By induction assumption, there is an execution of `REF(`$\mathcal{C}, sn_1$`)` that returns b. Therefore, there exists an execution of `REF(`$\mathcal{C}, a_0's\ m_1$`)` that chooses c_1 and eventually returns b.

Continuing the induction step. When sn is a compound name, let sn be “$a_0's\ m_1's\ m_2's \ldots m_q$,” where $q > 1$; consider the sequence of names resulted from the reduction: $sn_0 = sn, sn_1, \ldots, sn_p = b$, where sn_i is the result of applying the certificate c_i to sn_{i-1}. Because each reduction step only changes the first two symbols in a name, the only way to shorten a name is to replace the first two symbols with one principal, and the name sn is finally reduced to a single principal, then there must exist a point at which the name $(a_0's\ m_1)$ is reduced to a principal and “$m_2's \ldots m_q$” remains unchanged. Let this point be reduction step t, then $1 <= t < p$ and $sn_t = a_t's\ \ m_2's \ldots m_p$. Therefore, the certificates “$c_1, c_2, \ldots, c_t$” reduce $(a_0's\ m_1)$ to a_t

and the certificates "$c_{t+1}, \ldots, c_p$" reduce $(a'_t s\ m'_2 s \ldots m_q)$ to b. By induction assumption, there is an execution of `REF` $(\mathcal{C},\ a'_0 s\ \ m_1)$ that returns a_t and there is an execution of `REF` $(\mathcal{C}, a'_t s\ m'_2 s \ldots m_q)$ that returns b. Therefore, there exists an execution of `REF` $(\mathcal{C}, sn)$ that recursively calls `REF` $(\mathcal{C}, a'_0 s\ m_1)$ and `REF` $(\mathcal{C}, a'_t s\ m'_2 s \ldots m_q)$ and returns b.

We now prove the if part. Suppose that there is an execution of `REF` $(\mathcal{C}, sn)$ that returns b. Induce on the number of calls to `REF` during the execution; let the number be p. If p is one, sn must be b, so sn is trivially reducible to b. Recall that sn is a SPKI name, *i.e.*, a principal followed by a list of local identifiers, so sn can not has the form $(a'_0 s\ b)$, it has to be a single principal b. If p 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 sn has the form $(a'_0 s\ \ m_1)$. Let c_1 be the certificate chosen on line 5-6 and sn_1 be the subject of c_1. Because the call on line 7 returns b with $p - 1$ calls of `REF`, by induction assumption, there exists a sequence of name certs that reduces sn_1 to b. Therefore, c_1 followed by this sequence of name certs reduce sn to b. If the first recursive call happens on line 10, then sn has the form $(a'_0 s\ \ m'_1 s \ldots m_q)$, where $q > 1$. The call `REF` $(\mathcal{C}, a'_0 s\ m_1)$ (on line 10) returns a principal a_t with less than p calls of `REF`, and the call `REF` $(\mathcal{C}, a'_t s\ m'_2 s \ldots m_q)$ (on line 11) returns b within less than p calls of `REF`. By induction assumption, there exist one sequence of certificates that reduce $a'_0 s\ m_1$ to a_t and another sequence of certificates that reduce $a'_t s\ m'_2 s \ldots m_q$ to b. Concatenating two sequences together, they reduce sn to b. ∎

Proposition 3 Equivalence of REF and $\mathcal{P}_4$: *A SDSI name "sn" is resolvable to a principal "b" given a set of name certs $\mathcal{C}$ if and only if the name "sn" contains the principal "b" given $\mathcal{C}$.*

Proof. First, let us prove the if part. If there is a proof sequence for $contains(sn, b)$, induce on the length of the sequence. When the length is one, either Globality rule or Self-containing rule is used; in either case, sn is trivially resolvable to b. Otherwise, consider the rule that is used in the last step. Again, sn is trivially resolvable to b if it is either the Self-containing rule or the Globality rule. If it is the Superset rule, then sn is of the form $(a'_0 s\ \ m_1)$, there is a certificate in $\mathcal{C}$ that is represented by $includes([a_0, m_1], sn_1)$, and the atom $contains(sn_1, b)$ appears earlier in the proof sequence. By induction assumption, there is an execution of `REF` $(\mathcal{C},\ sn_1)$ that returns b. Therefore, there is an execution of `REF` $(\mathcal{C},\ a'_0 s\ m_1)$ that goes to line 5-7 and returns b. If the last step uses the Linking rule, then sn is of the form $(a'_0 s\ \ m'_1 s\ \ m'_2 s \ldots m_q)$; both $contains([a_0, m_1], a_1)$ and $contains([a_1, m_2, \ldots, m_q], b)$ appear earlier in the sequence. By induction assumption, there exists an execution of `REF` $(\mathcal{C}, a'_0 s\ \ m_1)$ that returns a_1 and there exists an execution of `REF` $(\mathcal{C}, a'_1 s\ \ \ m'_2 s \ldots m_q)$ that returns b. Therefore, there exists an execution of `REF` $(\mathcal{C}, a'_0 s\ m'_1 s\ m'_2 s \ldots m_q)$ that goes through lines 10 and 11 and returns b.

We now prove the only if part. Suppose that there is an execution of `REF` $(\mathcal{C},\ \ sn)$ that returns b. Do induction on the number p of all recursive calls of `REF` in the execution. If p is one, then sn is either b or $(a'_1 s\ \ b)$. Either the Self-containing rule or the Globality rule will prove $contains(sn, b)$. If p 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 sn has the form $(a'_0 s\ m_1)$. Let c_1 be the certificate chosen on lines 5 and 6 and sn_1 be the subject of c_1. Because the call on line 7 returns b with $t - 1$ recursive calls, by induction assumption, there exist a proof sequence for $contains(sn_1, b)$. This sequence followed by $includes([a_0, m_1], sn_1)$ and $contains([a_0, m_1], b)$ is a proof sequence for $contains(sn, b)$; the last step uses the Superset rule. If the first recursive call happens on line 10, then sn has the form $(a'_0 s\ \ m'_1 s \ldots m_q)$, where $q > 1$. Then the call `REF` $(\mathcal{C}, a'_0 s\ \ m_1)$ (on line 10) returns a principal a_t within less than p recursive calls, and the call `REF` $(\mathcal{C}, a'_t s\ \ m'_2 s \ldots m_q)$ (on line 11) returns b within less than p calls. By induction assumption, there exist one proof sequence for $contains([a_0, m_1], a_t)$ and one for $contains([a_t, m_2, \ldots, m_q], b)$. Concatenating them together and add $contains([a_1, m_1, m_2, \ldots, m_q], b)$ to the end is a proof sequence for $contains(sn, b)$. The last step uses the Linking rule. ∎