How to Resolve SDSI Names Without Closure

Sameer Ajmani

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139, USA
ajmani@lcs.mit.edu
First Draft: June 6, 2002
Last Modified: September 11, 2002

1 Introduction

Previous work on SDSI name resolution and authorization checking requires that the set of certificates be closed under name-reduction [1, 2, 4]. This paper presents two new algorithms that support efficient and complete SDSI name resolution without requiring closure over the set of certificates. This is particularly important for large, distributed certificate sets where calculating the full closure may be prohibitively expensive (in terms of time, storage, and/or communication).

The algorithms work by fetching the required certificates from a directory and then calculating the closure locally over those certificates. Our main contributions are (1) determining which certificates must be fetched to derive a result, (2) determining which operations the certificate directory must support, and (3) showing that it is possible to "reverse" name resolution; that is, given a key, we can determine the set of names that resolve to it. This is particularly valuable for authorization checks, since the number of names per key is usually much smaller than the number of keys per name.

The first algorithm takes a name and returns its value (a set of keys). This is useful for determining the set of keys that can access a protected resource. The second algorithm takes a key and returns the set of local names whose value contains that key. This is useful for determining whether a particular key can access a protected resource.

Both algorithms can be combined with SPKI authorization certificates to support efficient authorization checks without closure. We have not yet worked out the details for supporting threshold subjects, but we expect this to be straightforward.

2 Data Model

We define the following data types:

- A *principal* is a public key.
- A *name* is a 2-tuple: $\langle issuer, names \rangle$. *Issuer* is the principal that defines this name; i.e., *issuer* identifies the namespace in which this name is defined. *Names* is a non-empty sequence of strings. If *names* contains exactly one string, this is a *local name* that is defined by this *issuer*. Otherwise, this is an *extended name* that is resolved recursively through a series of namespaces. The *value* of a name is a set of principals; this value is defined by certificates signed by the issuer.
- A *subject* is either a principal or a name.
- A *certificate* is a 2-tuple: $\langle name, subject \rangle$. *Name* is a local name. A certificate states that the value of its *name* is a set that includes the value of its *subject*.
- A *signature* is a cryptographic signature.

- A *sequence* is an ordered sequence of ⟨*certificate*, *signature*⟩ pairs, where each *signature* is the certificate *issuer*'s cryptographic signature of the *certificate*.
- A *proof* is a 3-tuple: ⟨*name*, *subject*, *sequence*⟩. A proof is essentially a name-subject binding coupled with the evidence for that binding (certificates and signatures). A verifier can check a proof by treating each certificate as a "rewrite rule" and checking whether the *sequence* of rules rewrites the proof's *name* to its *subject* [2]. Thus, verification takes linear time in the length of the proof's *sequence*.

3 Basic Closure Algorithm

We begin by showing how to resolve SDSI names using the name-reduction closure over a certificate set [2]. The result of a name resolution is a set of proofs that bind the desired name to principals. This algorithm is not designed to work in distributed systems, although it can be adapted to that environment using a distributed hash table [1]. Later sections will extend this algorithm to work with any certificate directory that provides the required lookup services.

Figure 1 gives the pseudocode for this algorithm. The core routine is **compose**, which composes two proofs. Composition creates a new proof that represents a conclusion drawn from the two input proofs. The new proof contains the proper *sequence* of certificates to support this conclusion. Only certain pairs of proofs meet the precondition of **compose**; we call such proof pairs *compatible*. **Compose** does not commute.

We illustrate **compose** with an example. Suppose we have two compatible proofs: (K_{MIT} staff $\longrightarrow$ K_{MIT} faculty) and (K_{MIT} faculty $\longrightarrow$ K_{Rivest}). The first says that "K_{MIT} staff" is a set that contains "K_{MIT} faculty"; the second says that "K_{MIT} faculty" contains the principal K_{Rivest}. Given these two proofs, **compose** returns the proof (K_{MIT} staff $\longrightarrow$ K_{Rivest}). This new proof contains the concatenation of the *sequences* of

```
insert(certificate c)
   proof p
   p.name ← "K n" (the name bound by c)
   p.subject ← c's subject
   p.sequence ← c, c_{K^-1}
   insert(p)

insert(proof p)
   if (check[p.name, p.subject] is empty)
      check[p.name, p.subject] ← p
      if (p.subject is a name)
         compatible[prefix(p.subject)]
            ← compatible[prefix(p.subject)] ∪ {p}
         set ← value[prefix(p.subject)]
         for each p' ∈ set
            insert(compose(p,p'))
      else
         value[p.name] ← value[p.name] ∪ {p}
         set ← compatible[p.name]
         for each p' ∈ set
            insert(compose(p', p))

// returns the local name that begins n
prefix(name n)
   return name⟨n.issuer, n.names[0]⟩

// requires p1.subject = p2.name · X
// for some (possibly empty) sequence of strings X
// returns the composed proof p
compose(proof p1, proof p2)
   p.name ← p1.name
   p.subject ← p2.subject · X
   p.sequence ← p1.sequence · p2.sequence
   return p
```

Figure 1: *Basic closure algorithm*

certificates of the two input proofs, which is exactly what the verifier needs to check this proof.

The closure algorithm uses three local hash tables [3]:

check Maps a ⟨*name*, *subject*⟩ pair to a proof with the same *name* and *subject*. Used to avoid creating duplicate proofs and to terminate loops.

value Maps a local name n to a set of proofs whose *name* is n and whose *subject* is a principal. That is, value maps a name to proofs that resolve that name.

compatible Maps a local name n to a set of proofs whose subject is a name that starts with n. Thus, proofs whose *name* is n are compatible with proofs in compatible[n].

The **insert** routine is designed to maintain a single invariant: if proofs p_1 and $p2$ are compatible and if **insert**(p_1) and **insert**(p_2) occur (in any order), then **insert**(**compose**(p_1, p_2)) occurs before one of the parent insertions completes. This property guarantees that when all calls to **insert** complete, if there is some sequence of proof compositions that yields a proof ($n \longrightarrow k$) for some local name n and some principal k, then value[n] will contain that proof. Given this property and a set of certificates, one can easily determine the value of a local name n by calling **insert** on each certificate and then looking up value[n].

Inserting a proof p works as follows: if p has already been inserted, return immediately; otherwise, add p to the check table to prevent future insertions of p. If p's subject is a name, then add p to the compatible table, look up compatible proofs in the value table, and recursively insert the compositions of p with the compatible proofs. Otherwise, p's subject is a principal (or other non-name object), so add p to the value table, look up compatible proofs in the compatible table, and recursively insert the compositions.

This algorithm has two major drawbacks. First, this algorithm takes a long time to run over a large set of certificates ($O(N^3)$ for N certificates), especially if the set is distributed over a network. Second, this algorithm generates all possible proofs given the certificate set, even though the user may only need a particular proof or set of proofs. This wastes both time and storage.

```
// returns all reducing proofs whose name is n
resolve(name n)
   load-value(n)
   return value[n]

insert(proof p)
   if (check[p.name, p.subject] is empty)
      check[p.name, p.subject] ← p
      if (p.subject is a name)
         compatible[prefix(p.subject)]
            ← compatible[prefix(p.subject)] ∪ {p}
▷        load-value(prefix(p.subject))
         set ← value[prefix(p.subject)]
         for each p′ ∈ set
            insert(compose(p,p′))
      else
         value[p.name] ← value[p.name] ∪ {p}
         set ← compatible[p.name]
         for each p′ ∈ set
            insert(compose(p′, p))

load-value(name n)
   if n ∉ loaded-value
      loaded-value ← loaded-value ∪ {n}
      set ← fetch proofs whose name is n
      for each p ∈ set
         insert(p)
```

Figure 2: *Name resolution algorithm*

4 Name Resolution

We improve on the basic closure algorithm with our name resolution algorithm in Figure 2. This algorithm does not maintain closure over the entire certificate set; instead, it fetches only the certificates needed to resolve the requested name and calculates closure locally over those certificates. The only requirement is that the certificate directory be capable of returning all proofs whose *name* is a particular local name.

The change to the body of **insert** is marked by ▷. The **load-value** routine accesses the certificate directory and inserts all proofs that resolve the given name. **Load-value** uses a local cache, loaded-value, to ensure that the certificate directory is only accessed once per name.[1] The rest of the **insert** routine runs the compositions needed to calculate closure over the fetched certificates (possibly requiring additional fetches). Thus, to determine the value of a local name, we call **load-value** on that name and then look up value$[n]$.

The main problem with **resolve** is that it might explore large areas of the certificate graph. **Resolve** starts from the desired name and does a depth-first search. If that name or a name it depends on is bound to a large set of keys, the search may take a long time. However, **resolve** is careful not to repeat work, so the total amount of work done is proportional to the size of the result. Thus, **resolve** is useful when one wants to determine the complete value of a name, for example, to determine which principals can access a protected resource.

5 Unresolution

The previous algorithm can be extremely inefficient for checking whether a particular principal can access a resource. For example, suppose a resource is only accessible by principals in the group "K_{USA} citizens". To determine whether a key K can access this resource, one must first determine the value of "K_{USA} citizens", which contains millions of principals!

Instead, it is far more efficient to determine which names' values contain K. For most principals, this is a small set of names.[2] One way to do this is to calculate closure over the set of certificates then locate those proofs whose *subject* is K. However, calculating closure over a large, distributed certificate set may be impractical. Instead, we would like to fetch only the required certificates from a certificate directory and calculate closure locally over these. We call this process "unresolution" of a principal; Figure 3 presents our algorithm.

This algorithm uses a fourth local hash table, reverse, that maps a principal to proofs whose subject is that principal. This algorithm requires that the certificate directory support two kinds of lookups. First, given a principal k, the directory should return proofs whose *subject* is k. Second, given a name n, the directory should return proofs whose subject is a name that starts with n. Two caches, loaded-reverse and loaded-compatible, ensure we only access the certificate directory once for a given query.

The differences between this version of **insert** and the original are marked by ▷. This reverse mapping alone is enough to discover those names that are bound directly to a given principal. However, this will not find extended names that are bound to that principal. To do so, this algorithm recursively searches through each name's issuer using **load-reverse**. This discovers any extended names that could resolve to the principal; the calls to **load-compatible** ensure that the necessary compositions occur to find proofs for those extended names.

We clarify this process with an example. Suppose we have the following set of proofs in our certificate directory:

$$K_{MIT} \text{ staff} \longrightarrow K_{MIT} \text{ faculty assistant} \quad (1)$$
$$K_{MIT} \text{ faculty} \longrightarrow K_{Rivest} \quad (2)$$
$$K_{Rivest} \text{ assistant} \longrightarrow K_{Be} \quad (3)$$

If we **unresolve** K_{Be}, we immediately **load-reverse** K_{Be} and insert (3). This in turn causes us to **load-reverse** K_{Rivest}. We then insert (2), which causes us to **load-compatible** proofs whose subject starts with "K_{MIT} faculty". This includes (1), so it is also inserted. This insertion causes (1) to be composed with (2), which is in our local value table. The resulting proof, (K_{MIT} staff $\longrightarrow$ K_{Rivest} assistant), is recursively inserted and so is composed with (3), which is also in our local value table. This final result, (K_{MIT} staff $\longrightarrow$ K_{Be}), is inserted.

[1] One should, of course, periodically refresh the cache. This involves purging expired or revoked proofs and adding any new ones using **insert**.

[2] An exception might be a well-known individual, such as the US president, who would be identified by local names in many different namespaces

```
// returns all reducing proofs whose subject is k
unresolve(principal k)
   load-reverse(k)
   return reverse[k]

insert(proof p)
   if (check[p.name, p.subject] is empty)
      check[p.name, p.subject] ← p
      if (p.subject is a name)
         compatible[prefix(p.subject)]
            ← compatible[prefix(p.subject)] ∪ {p}
         set ← value[prefix(p.subject)]
         for each p′ ∈ set
            insert(compose(p,p′))
      else
         value[p.name] ← value[p.name] ∪ {p}
▷        reverse[p.subject] ← reverse[p.subject] ∪ {p}
▷        load-compatible(p.name)
         set ← compatible[p.name]
         for each p′ ∈ set
            insert(compose(p′, p))
▷        load-reverse(p.name.issuer)

load-compatible(name n)
   if n ∉ loaded-compatible
      loaded-compatible ← loaded-compatible ∪ {n}
      set ← fetch proofs whose subject starts with n
      for each p ∈ set
         insert(p)

load-reverse(subject s)
   if s ∉ loaded-reverse
      loaded-reverse ← loaded-reverse ∪ {s}
      set ← fetch proofs whose subject is s
      for each p ∈ set
         insert(p)
```

Figure 3: *Unresolution algorithm*

Thus, reverse$[K_{Be}]$ contains "K_{MIT} staff" and "K_{Rivest} assistant" (from the original insertion of (3)).

The main problem with **unresolve** is that it must recursively explore every path of certificates that leads to a particular key. Given the "six degrees of separation" phenomenon, this could lead to a very large fraction of the set of principals! The simplest defense against this problem is to truncate the search at a certain depth, since longer certificate chains are less trustworthy than shorter ones. An analysis of real certificate networks would provide better insight into this problem.

6 Extended Names

We have defined both **resolve** and **unresolve** in terms of local names: **resolve** finds all keys in a local name's value, and **unresolve** finds all local names that resolve to a key. In both cases, we may want to use extended names instead. This is straightforward using dummy names.

To **resolve** and extended name n, create a dummy proof p whose *name* is some dummy name d, whose *subject* is n, and whose *sequence* is empty. **Insert** p, then **resolve** d. The resulting proofs are the same as those that resolve n.

To determine whether the value of n contains some key k, construct proof p as above, **insert** p, then **unresolve** k. If the resulting proofs include one that resolves d to k, then the same proof resolves n to k.

7 Directed Search

We have defined both **resolve** and **unresolve** to return sets of proofs. In access control, the user typically wants just one proof: one that shows that particular name resolves to a particular key. Given both the desired anme and key, we can make the above algorithms more efficient. Specifically, each time **insert**(p) is called, it checks whether p proves the desired name-key binding. If so, it returns p immediately and aborts all further processing. A simple way to

implement this is to "return" p by throwing an exception.

8 Authorization Proofs

Each of our algorithms can be combined with authorization certificates to find authorization proofs. Suppose we want to determine which principals are authorized to access a particular resource. For each entry on that resource's ACL, we perform a standard graph search using authorization certificates [2, 4]. If we encounter a name in the subject of an authorization certificate or an ACL entry, we process that name using **resolve**. This maps the name to a set of keys, allowing us to continue our graph search.

Suppose instead we want to determine whether a particular principal can access a given resource. We attempt a reverse graph search using authorization certificates starting from the desired principal. In addition, we **unresolve** each principal encountered during the search. Then, for each such principal k, $\mathsf{reverse}[k]$ contains authorization certificates whose subject is a name that resolves to k. We can continue our reverse graph search through these authorization certificates.

References

[1] S. Ajmani, D. Clarke, C.-H. Moh, and S. Richman. ConChord: Cooperative SDSI certificate storage and name resolution. In *Proceedings International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.

[2] D. Clarke, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 2001.

[3] C. M. Ellison and D. E. Clarke. High speed TUPLE reduction. Memo, Intel, 1999.

[4] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. In *Proc. 8th ACM CCS*, Nov. 2001.