SDSI – A Simple Distributed Security Infrastructure

Ronald L. Rivest
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rivest@theory.lcs.mit.edu

Butler Lampson*
Microsoft Corporation
blampson@microsoft.com

April 30, 1996

Abstract

We propose a new distributed security infrastructure, called SDSI (pronounced "Sudsy"). SDSI combines a simple public-key infrastructure design with a means of defining groups and issuing group-membership certificates. SDSI's groups provides simple, clear terminology for defining access-control lists and security policies. SDSI's design emphasizes linked local name spaces rather than a hierarchical global name space.

1 Introduction

This paper was motivated by the perception that the existing proposals for a public-key infrastructure (such as X.509-based schemes that require global certificate hierarchies [3]) are both excessively complex and incomplete. Their complexity arises from a dependence on global name spaces and an attempt to formalize too many things. Their incompleteness can be immediately perceived if one tries to define a security policy (e.g. write a ACL) based on the scheme.

We were also motivated by similar corresponding design efforts, such as that of the IETF "SPKI" (Simple Public-Key Infrastructure) working group (in particular, Carl Ellison's work), the work of Blaze, Feigenbaum, and Lacy [1], and work within the World-Wide Web Consortium's security group. Our proposal borrows freely from these sources, and has benefitted greatly from them.

Our goal is to provide ideas and techniques that facilitate the construction of secure systems. Issues regarding the liabilities and/or the legal interpretation of certificates are not addressed here. Because we feel security is enhanced by simple clear data structures, we emphasize clarity and readability at the expense of economical encodings, although SDSI includes means for efficiently representing its data structures.

2 Overview

We first give an "executive summary" of our proposal here, and then flesh it out in the following sections. A detailed specification suitable for as an implementation guide will appear in a companion paper.

*Supported by Microsoft

Principals are public keys. Our system is "key-centric": SDSI principals *are* public digital signature verification keys. These public keys are central; everything is based around them. The notion of an "individual" (e.g. person, corporation, process, or machine) is not required. Of course, such individuals will actually control the associated private keys, so that the public/private keys can be viewed as "proxies" for those individuals. In this paper, one should think of a "principal" as a public key, and concurrently as "one who speaks" (by signing statements that can be verified with that public key). Each principal is represented by a data structure that can be passed around, such as:

```
( Principal:
  ( Public-Key:
    ( Algorithm: RSA-with-SHA-1 )
    ( N: =Gt802Tbz9HKm067= )
    ( E: #11 ) ) )
```

(other optional fields not shown) so a principal can also be the "value" of some name. Each principal may also have an associated name-space, servers, database, etc., as we shall see.

Egalitarian design—no global hierarchy necessary. Our proposal is egalitarian: each principal can make (signed) statements and requests on the same basis as any other principal. No hierarchical global infrastructure is required. Of course, in practice some principals would be more important than others, and SDSI allows for some principals to have special status as "distinguished roots," allowing SDSI to accomodate "global names." But that is for convenience rather than necessity.

Simple data structures. SDSI objects are represented by "S-expressions" (octet-strings and lists of simpler S-expressions) having a clean human-readable ASCII representation. "Presentation hints," based on MIME content-types, enable octet-strings to be presented in italics, or as Unicode characters, or as photos, etc. There are also ways to transmit S-expressions efficiently.

Flexible signatures. SDSI signatures are exceptionally flexible: signatures may be appended to the objects being signed, or they may be detached from the signed objects. Objects may be co-signed by several signers. Signers may sign on behalf of someone else. Signatures may contain collections of relevant certificates. Signatures may have expiration dates, and may also require periodic re-confirmation.

Identity certificates have human-readable content. The term "certificate" is often used to denote a digitally signed statement. The best-known type of certificate is the "identity certificate" that attempts to assert a binding between an individual and his public key.

Identity certificates are often problematic, because it can be difficult to specify a name or other attributes that uniquely identify the desired individual. We feel that because of this difficulty, identity certificates must typically in the end be examined by people, to see if the name and other attributes given are consistent with the attributes known to the human reader. As a typical example, consider the problem of a security administrator who is trying to decide if a given certificate should be trusted as establishing the desired binding between some public key and some individual. We feel that human judgement is a necessary component in this process, and thus we feel that the function of an identity certificate can best be served by having the individual described by some free-form text (accompanied perhaps by a photo, and perhaps including some descriptive text describing how the given information was verified). For this reason, SDSI identity certificates may contain as little as a public key and some free-form text about the individual.

Certificates also give name/value bindings and assert membership. SDSI certificates can also be used to bind a (local) name to a value (typically a principal), and to assert that a given principal belongs to some group. The three certificate types (identity certificate, name/value certicate, and membership certificate) have similar syntax and procedures; indeed, sometimes a single SDSI certificate can perform more than one function. SDSI certificates are also "extensible;" additional application-dependent data can be included in a certificate.

Each principal is a "certification authority." In SDSI, certificates can be created and signed by anyone: everyone can be a "CA." Which policies and procedures (if any) a principal follows when issuing certificates is up to that principal to choose: he may declare that he is following some industry-standard procedure, or he may issue certificates arbitrarily or capriciously.

Local name spaces. Each principal can create his own local names with which he can refer to other principals. These local names arbitrarily chosen; they may be derived from nicknames, email addresses, account numbers, etc.:

```
jim
"Bob Jones, Jr."
account-314568901387
WebCo-Vice-President-John-Smith
joe@penguin.lcs.mit.edu
( Accounting Bob-Smith )
```

There is no fixed "global" name space giving a unique name for principals. The principal you call `alice-smith` may be different from the principal I call `alice-smith`. An exception is made for a small set of "distinguished root" principals, whom everyone calls by the same name.

Manual process for creating identity certificates We feel that the process of accepting another individual's public key(s) and associating it with a local name is very important, and that it must therefore be a *manual* process. Judgement is required to evaluate the proposed evidence for the public-key's owner, and to choose a meaningful local name. A typical user will not have to perform this operation very often (perhaps 5-20 times), as we shall see, because of the way name spaces can be linked.

On-line Internet orientation. We assume that principals who issue certificates can provide on-line Internet service, or can arrange to provide such via a designated server. A SDSI principal may sign certificates off-line, and have a server distribute them upon request. Having such on-line capability permits considerable simplifications—for example, we eliminate "certificate revocation lists" in favor of on-line "reconfirmation."

Linked local name spaces. SDSI does not utilize a global name space, but rather provides means for conveniently linking local name spaces. Each principal can "export" his name/value bindings to others, by issuing name/value certificates. Thus, if my local name `bob` refers to some principal, then I can refer to the principal that `bob` calls `alice` as

```
( ref: bob alice ) .
```

We suggest that the user interface supply a bit of syntactic sugar to represent this as

```
bob's alice
```

Bob exports his binding by issuing a signed name/value certificate binding his local name `alice` to that particular principal. Bob's server can distribute this certificate on demand.

It is not necessary to have a symbolic reference as the first argument to `ref:`; one can have an expression such as

(`ref:` *principal* `alice`)

where *principal* is an explicit principal (i.e., a public key, etc.). The syntactically sugared version of the above would be:

principal`'s alice` .

However, we generally prefer the symbolic form, for clarity.

One can also have longer references, such as

```
bob's alice's mother ,
```

which is the syntactically sugared version of

```
( ref: bob alice mother ) .
```

This reference is well-defined (for me) if I have bound `bob` to some principal, who in turn has bound `alice` to some second principal, who in turn has bound `mother` to some third principal. Other examples of extended references (with syntactic sugar) are:

```
Visa's account-314568901387
mit's lcs's rivest
cmu's eecs-dept's search-committee's chairman
GE's lighting-division's vp-of-marketing's secretary's assistant
```

As a detail, we note that `( ref: bob )` is equivalent to `bob`, since the first argument to `ref` is always interpreted in the current name-space.

One can also give "symbolic definitions." For example, my local name `tim` might denote `mit's Tim-Black`. This means that my `tim` is defined to be the same principal as the one `mit` refers to as `Tim-Black`. If MIT changes the principal it calls `Tim-Black` then the principal I call `tim` changes as well. The pragmatics of how this works are described below.

To ensure that the object referenced is the intended one, one can wrap the `ref:` in an `Assert-Hash:`

```
( Assert-Hash: ( ref: Microsoft employee-list )
               ( SHA-1 =hP+73N2Zr50/89jpio== )
```

This is equivalent to `ref:`, except that the retrieved value is hashed and compared to the given hash. An error occurs if they are not equal.

Accomodation for "standard roots" and global name spaces: We recognize that there are likely to be a set of standard "distinguished root" principals that are "universally" recognized. These principals have SDSI reserved names ending with double exclamation marks (`!!`):

```
VeriSign!!
IAPR!!
USPS!!
DNS!!
```

There will be very few of these distinguished roots, and *their names are bound to the same principal in every name space.* This is arranged with suitable procedures (appropriate publicity, manual installation, cross-checking, etc.); we do not try to describe this in more detail. This gives SDSI access to "standard" name spaces:

```
VeriSign!!'s MicroSoft's Encarta-Division's Susan-Smith
DNS!!'s edu's mit's lcs's theory's Silvio-Micali
USPS!!'s USA's DOD's DCI
IAPR!!'s PCA-commerce's DEC's SRC's abadi
VeriSign!!'s Visa's account-234156742
```

Each of these should be viewed as *global* names that evaluates to the *same* principal in every name space, since the first name (e.g. `VeriSign!!`) always evaluates to the same principal, and all of the subsequent names are relative.

Although SDSI provides for distinguished roots within a set of linked name spaces, this does not imply that each principal has a *unique* "global name." A principal will have multiple global names if there are multiple paths from distinguished roots to that principal. Thus,

```
VeriSign!!'s MicroSoft's CEO
DNS!!'s com's microsoft's "Bill Gates"
```

may refer to the same principal.

DNS names have a special status. By special dispensation from its designers, SDSI includes custom treatment for DNS (Internet email) names, so that

```
Bob.Smith@penguin.microsoft.com
```

is equivalent to:

```
DNS!!'s com's microsoft's penguin's Bob.Smith
```

—that is, to:

```
( ref: DNS!! com microsoft penguin Bob.Smith ) .
```

How this works is described later on.

Groups. A SDSI *group* is typically a set of principals. (The full notion is more general, and easily extends to sets of programs or web pages or other things.) Each group has a name and a set of members. The name is local to some principal, who is the "owner" of the group. The owner is the only one who may change the group definition. The definition may explicitly list the group's members, or define the group in terms of other groups (which may even belong to someone else).

There are good reasons why "groups" are a fundamental SDSI notion. For describing who is authorized to access certain data or to perform certain operations, it is usually simplest to define the group of authorized principals in one step, and then to place the group name on the appropriate access-control list(s) as a second step. This provides efficiency and reliability when the same group of principals is authorized on many different ACL's. The group definition can be updated with a single modification without having to update many ACL's. Also, the group can be given a meaningful name, so that auditing group definitions and ACL's becomes simpler. For writing clear security implementations, it is extremely useful to refer to an authorized group of principals by an appropriate group name. Each principal can define his own groups, and export his definitions to others. Access-control lists for some resources might contain the following group names, for example.

```
friends
mit's biology-department's faculty
Massachusetts's DMV's drivers-96
john's search-committee-members
USA's over-18
```

One can define a groups by listing its members:

```
( Group: Tom Sam "Bill Gates" )
```

or by giving an algebraic expression in terms of the other groups or principals:

```
( Group: ( OR: faculty staff Bob-Smith ) )
```

We feel that the auditability of ACL's specified in terms of simple group names, and the auditability of the group definitions, is very important when trying to implement an appropriate security policy.

Group membership assertions can play the roles of credentials, licenses, or ordinary paper certificates.

- An organization such as ACM can create a group `members` that I might refer to as `acm's members`. A statement signed by `acm` might say that a given principal (i.e. a particular public key, not a name for that key) is a member of the group `members`. The "meaning" of this statement is entirely up to ACM, since it is ACM's prerogative to define its groups any way it pleases. The definition of the group `members`, and the assertion that some principal is a member of it, are things that ACM may export.
- The Department of Motor Vehicles may issue certificates asserting that (the individual controlling) a specified public-key is over 18 and has a valid Massachusetts driver's license for 1996 by issuing a signed statement that the principal is a member of DMV's group called `adult-drivers-1996`.

Each such certificate also has an expiration date and other information. Membership certificates are very useful for dealing with large groups, where it is not reasonable to export the entire group definition.

Clean support for "roles". Access control is often conveniently organized around "roles" (see Sandhu et al. [6]). SDSI supports roles in two different ways.

In the first method, an individual creates a public-key for each role he plays. Bob may create new principals (public keys) and give them appropriate names:

```
faculty-role
bob-jones-travel-key
"FudgeCo President"
"Biology Department Search Committee Chair"
```

Bob can sign statements as the principal I call `bob`, or he can sign statements as the principal I would call `bob's faculty-role`, and so on. I can distinguish these cases, and recognize when Bob is acting in his faculty role, as opposed to acting as Bob.

A second method by which SDSI can support roles is by having a group created for each role. The owner of the group is the principal who decides who should occupy that role. For example, a hospital administrator create a group for each role. The principal who is currently acting in that role can be made a member of that group. A hospital administrator can implement the role of `head-nurse` by creating a group with that name, and defining its (sole) member to be the principal of the head nurse.

These two methods differ according to who controls who is acting in the role. Also, there is no public-key for a group, so that the second method has the slight disadvantage that the principal playing that role signs with his full authority, rather than with the authority restricted to the permissions of his role.

Delegation Certificates. SDSI includes the ability to delegate authority in two ways: by using groups, and by using "delegation certificates". A delegation certificate gives the delegee the authority to sign statements of certain types on behalf of the delegator. For example, a server might have a delegation certificate allowing him to sign membership certificates on behalf of the delegator.

This concludes our brief overview of SDSI.

Outline of the rest of the paper. The following sections give some suggested details.

- Section 3 describes the common underlying data structures and syntax for SDSI, including the S-expression data structure and their representation as ASCII strings, hash values of an S-expression, dates, numbers, compression techniques, and the representation of object and message types.
- Section 4 describes some standard SDSI data types: cryptographic keys, principals (made from a public signature verification key and an internet address), encrypted objects, and signed objects (in three formats)
- Section 5 explains how each principal establishes his own local name space and how he can use certificates to export name/value bindings to other parties.
- Section 6 discusses protocols, gives the basic "query" protocol, gives the "reconfimation" protocol, and discusses autocertificates (which can be viewed as the reply to a cryptographic "finger" operation).
- Section 7 explains the notion of a "group," and shows how to use groups to specify authorization, how to define a group, how to determine whether a given principal is a member of some group, using the group definition and any available credentials, how to query a server to find out if a principal is a member of some remotely-defined group, and how to write ACLs in a simple manner.
- Section 8 gives examples of some typical application scenarios.

A companion paper will give precise specifications. This paper contains many suggestive details, but a fully worked-out proposal would be much lengthier. Although the first "S" in SDSI stands for "simple," this proposal may appear complex. However, given the rather ambitious scope of its aims, we feel that it is remarkably simple.

3 Data structures and Syntax

3.1 S-expressions

This section presents a general syntax for representing octet-strings and lists, the data structures used by SDSI.

Our data structures are "S-expressions," which may be:

- An *octet-string*: a sequence of zero or more 8-bit bytes. Each octet string may be preceded by an associated "presentation hint," which is also a sequence of 8-bit bytes. Examples of octet strings are: `"Digital Soup Kitchen"`, `Tom-Hanks`. An example of a presentation hint is `[image/gif]`.

- A *list* of zero or more simpler S-expressions. An example of a list is

```
( Algorithm: RSA-with-SHA-1 ) .
```

3.1.1 ASCII ("external") representations

This section describes how SDSI data structures can be represented in US-ASCII for transmission or storage purposes.

Each S-expression has ASCII "external" representations for transmission or storage. Their readability may help diagnosis and auditing. These representations are not unique; one can choose maximum economy or maximum legibility. For high efficiency, there is a "verbatim" representation of octet strings, and short forms for common symbols (see Section 3.3).

SDSI representations use the US-ASCII character set; they may use alphanumeric characters, "special" printing characters:

```
~ ` ! @ # $ % ^ & * - _ + = { } ; : ' " < > [ ] , . / ?  |
```

and whitespace (non-printing) characters (blanks, tabs, newlines, carriage-returns, etc). SDSI is case-sensitive. In general, the amount of whitespace between portions of the representation, and their arrangement on different lines, is immaterial. We show nicely indented structures for readability only; the indentation is not significant, and parentheses are used for grouping.

Octet-strings. There are six external representations for variable-length octet-strings. There is no limit on the length of an octet string.

- *Hexadecimal* (e.g. `#45facd57`); this is a sharp sign (`#`) followed by an even number of hex digits.

- *Decimal* (e.g. `45`), representing a non-negative integer less than 2^{15}. This denotes the shortest octet-string representing the given number in two's complement form.

- *Base-64* (e.g. `=Ac0+Thj390==`); this is an equals sign (`=`) followed by the base-64 encoding as defined in RFC-1521. This techniques encodes each triple of octets with a block of four characters from the 64-character set `A-Z a-z 0-9 + /`. Octet-strings whose length is not a multiple of three have a representation whose last four-character block contains two or three base-64 characters and two or one equals signs as padding. The length of a base-64 encoding is thus always a multiple of four.

- A *token* (e.g. `athena-server`) is a sequence of printable (non-whitespace) characters other than parentheses, single or double quotes, square brackets, or hash marks, which is not the decimal representation of an integer, and which does not end with a minus sign. A token that begins with a special character is called a "special token;" SDSI gives special treatment to some special tokens; they may not represent octet-strings at all, or may represent a different octet-string.

- *Quoted strings* (e.g. `"Mary P. O'Connor"`) represent octet strings containing blanks and printable characters other than double-quotes. On input, a consecutive sequence of whitespace characters within the quotes is converted to a single blank. On output, a single blank may be converted to an arbitrary sequence of whitespace characters (e.g. carriage returns may be freely added at word breaks—these are "word-wrap strings").

- *Verbatim octet-strings*, preceded by a hexadecimal length and a colon (e.g. `#03:^%-`), are used as an escape mechanism for transmitting large octet-strings most efficiently, at the cost of abandoning ASCII and human readability. The length is always given in the most compact form possible (no unnecessary leading zero octets).

An octet-string may be written as a sequence of "fragments" by following each fragment but the last with a minus sign. Thus `#0123- "b"- #01:c- d- #0876` is the same as `#01236263640876`. (Note that ASCII `"b"` is hexadecimal `#62`.) A long octet string can thus be fragmented for robust email transmission.

The various ways of expressing an octet-string are fully equivalent, as in the following example:

```
abc  "abc"  #616263  =YWRj  #03:abc   6382179  a- b- "c"
```

Octet-strings are used for a variety of purposes:

- Object type names, such as `Cert:` .

- Attribute names, such as `Date:` .

- Dates, such as `1996-02-14T11:46:05.046-0500`.

- Context-specific flags, such as `optional`.

- Names (such as `Mary-Jones` or `"Bob Smith"`) that may denote principals, groups, or other SDSI objects). A name need not be an octet-string; it may be an S-expression, such as `( ( Digital ) ( Accounting ) ( Robert Smith ) )` .

- Big numbers.

A *presentation hint* (discussed further below) is represented as an octet string surrounded by square brackets. An example is `[image/gif]`. A presentation hint is a modifier or tag for the following octet string; it does not itself represent an octet string, but rather describes how the following octet string might best be presented to the user.

Special tokens do not (necessarily) represent octet strings, but have special functions. At present, the only use SDSI makes of special tokens is for macros:

- Special tokens may represent macros, such as `*S` . Macros always start with a star `*` . A macro expands to an ordinary token. Note that `"*S"` is not a macro, and thus isn't expanded; a macro must be represented as a special token and not as some other form of an octet string.

S-expressions. An S-expression list can be represented in ASCII as a left parenthesis, an optional blank, the representations of the list elements separated by blanks, an optional final blank, and a right parenthesis. Blanks between list elements are optional if at least one element is itself a list—no blanks are needed between parentheses. On input, any sequence of white-space or non-printable characters not inside a verbatim octet-string is equivalent to a single blank. As an example of an S-expression:

```
( Get.Query:
   ( Template: ( Auto-Cert: ) )
   ( Signed:
     ( Object-Hash: ( SHA-1 #345678 ) )
     ( Date: 1996-02-14T11:46:05.046-0500 )
     ( Signature: #8dff4123 ) ) )
```

One can choose the representation of an S-expression according to the purpose being served:

- For maximum legibility, optional blanks are always used, and octet-strings are printed in the first manner that works of: token, quoted string, decimal (for numbers less than 2^{15}), or hexadecimal, and appropriate indentation is used.
- For maximum efficiency, optional blanks are never used, and the shortest possible representation of each octet string is chosen.
- For bullet-proof email transmission, fragmented base-64 encodings can be used, with line lengths limited to 76 characters.
- The "canonical" ASCII encoding of a SDSI object represents all octet-strings (including those within presentation hints) in verbatim mode. If an octet string has an associated presentation hint then the presentation hint precedes the octet-string and is separated from it by a single blank. Lists are represented with exactly one blank before and after each list item.

Presentation Hints. SDSI data structures may be sometimes be shown to people. In some cases, the rules above (such as those for "maximum legibility") may be inappropriate or inadequate. For example, some data may be text in Unicode, or may represent a photo. SDSI provides "presentation hints" for handling such cases.

These hints apply to the following S-expression, which must be an octet-string. (Each octet string may be preceded by at most one such presentation hint.) Presentation hints are octet-strings surrounded by square brackets, as in:

```
[charset=unicode-1-1] #34518976aabcde
```

specifies that the hexadecimal string `#345...` represents Unicode text (see RFC 1641). In general, we envision that one can use simple MIME Content-Types, such as

```
[text/richtext]
[image/gif]
[audio/basic]
[application/postscript]
[text/plain;charset=unicode-1-1]
```

The SDSI specification will say which presentation hints must be supported. Multipart MIME types are not supported. Other security-related types may be supported, such as

```
[application/X.509v3]
```

for including X.509v3 certificates within SDSI data structures. (Note that this is just a presentation hint for an octet string; SDSI does not parse or give any meaning to such a certificate, and such a certificate could be included as an octet string without any such presentation hint.)

Presentation hints are only hints. A presentation program may ignore the hint, and present the data in some default mode. If used, the hint itself should be suppressed in the presentation; if ignored, the hint should be displayed.

A hint may be placed before any octet string. For other purposes they are effectively invisible or ignored. They should be considered as a "tag" for the following octet string, and carried around with that string. However, they do participate in hash computations.

3.2 Numbers, Hash Values, and Dates

This section describes how (big) numbers, hash-values, and dates are represented as octet-strings.

Numbers. Integers are represented as an octet-string in two's complement notation with the most-significant byte first. A number may be of any length.

Hash Values. The hash value $H(S)$ of an S-expression S is obtained by applying a cryptographic hash algorithm, such as MD5 [5] or SHA-1 [4], to S's canonical ASCII representation. SDSI's default hash algorithm is SHA-1. A hash value is represented as a list containing the hash algorithm name and the hash value.

```
( SHA-1 =67adhNPs8Y+/Uy34NhWp77CvULm= )
```

Dates and Time. Dates/times are represented as UTC time to the millisecond, with a time-zone designation, as an octet-string in the following format:

```
yyyy-mm-ddThh.mm.ss.mmm-xxzz
1996-02-14T11:46:05.046-0500
```

for February 14th, 1996 at 11:46:05 AM in Eastern Standard Time. Here `yyyy` is the year, `mm` is the month, `dd` is the day, `hh` is the hour, `mm` is the minutes, `ss` is the seconds, `mmm` is milliseconds, and `xxzz` is the time zone expressed as a difference (either `+` or `-`) of local time to UTC time in hours (`xx`) and minutes (`zz`). This format follows the ISO 8601:1988 conventions. (However, the described format is required, with none of the variations possible within that standard.)

3.3 Compression

This section (which should be skipped at first reading) describes compression techniques for sending SDSI data structures efficiently.

SDSI has a fairly expansive, human-readable syntax. For greater economy in transmission, SDSI has a simple "macro" capability, as well as "verbatim" encodings for octet-strings.

Quote macro

The form `'`S, for any S-expression S, is equivalent to the form `( Quote:` S `)`.

Star macros. If a special token begins with a star, then it is looked up in a fixed published standard table of substitutions. Each such special token can have some token as its defined value. For example,

```
*C  ==> Cert:
*G  ==> Get.Query:
*P  ==> Principal:
*Sg ==> Signature:
...
```

All common SDSI tokens have such a "short form."

Within a protocol message, the reserved macros `*SENDER` and `*RECEIVER` refer to the principal for the sender and the principal for the receiver. Thus, `( ref: *SENDER bob )` refers to the sender's `bob`, while `( ref: *RECEIVER bob )` refers to receiver's `bob`.

Star macros are only used for transmission. They are used to compress a message before transmission, and to decompress upon receipt. They only exist "on the wire"; they are not used for storage or presentation, and are never present when a hash computation is performed.

3.4 Object and message types

This section describes how lists are used to represent SDSI objects, and a particularly important kind of object: messages.

Lists are used to represent most SDSI objects. The first element of the list gives the *type* of the object; the rest is a sequence of attribute/value pairs.

```
( type:
  ( Attribute1: value1 )
  ( Attribute2: value2a value2b value2c )
  ... )
```

We say that such an object has type `type:`. Object types and attribute names (also called field names) are representable as tokens that end with a colon. An attribute may have a list of values, as shown above for `Attribute2:`. The last pair is often a signature, with attribute `Signed:`, and a value containing the signature information. With the exception of the placement of `Signed:` attribute/value pairs (which occur last), the relative ordering of the other attribute/value pairs has no significance. (However, re-ordering attribute/value pairs yields a different SDSI object, and so a signature on the original object will not be valid if items have been re-ordered.)

A minor exception to the above format is the inclusion of one object directly in another. For example, an object of type `Public-Key:` can be directly included in a `Principal:`

```
( Principal:
   ( Public-Key: ( Algorithm: ... ) ... )
   ... )
```

rather than duplicating the object type as an attribute name:

```
( Principal:
   ( Public-Key: ( Public-Key: ( Algorithm: ... ) ... ) )
   ... )
```

Aside from the `Signed:` attribute, each attribute may appear at most once in an object. The `Signed:` attribute may appear multiple times. It is an error condition if any other attribute appears more than once, and it is important that SDSI programs check this condition and reject any objects that violate it. (There are other conditions that should also be checked, such as making sure that attributes with multiple values are allowed to have multiple values.)

A common attribute is `SDSI-version:`, whose value is the SDSI version number effective for that object:

```
  ( SDSI-version: 1.0 )
```

This paper describes SDSI version 1.0. Many SDSI objects will contain this attribute/value pair; we don't show this in our examples.

Messages are objects of certain types; the type is representable as a token of the form

```
protocol-name.message-type:
```

that identifies the type of the message transmitted during some two-party protocol. Each protocol has a unique name, such as `Get` or `Electronic-Payment`. Each protocol consists of a back-forth exchange of messages, and each message has its own name within the protocol, such as `Query`, `Reply`, or `Receipt`. Thus, a message type may be representable by a token such as:

```
Get.Reply:
Electronic-Payment.Receipt:
```

A commonly used attribute in a message might be the hash of the last message received from the other party:

```
   ( Your-Last-Message-Hash: ( SHA-1 =YnK82Ab+0m0i/uBx+9ikHjdF9NV= ) )
```

4 Standard SDSI Object Types

This section describes four standard SDSI objects types: cryptographic keys, principals, encrypted objects, and signed objects.

4.1 Keys and encryption parameters

Cryptographic keys are represented by an object that gives the key type (one of `Public-Key:`, `Private-Key:` or `Shared-Secret-Key:`), the algorithm specifier, and associated parameters.

```
( Public-Key:
   ( Algorithm: RSA-with-SHA-1 )
   ( N: =Hi7KugV013Tv978d00vCpQ== )
   ( E: #11 ) )
( Private-Key:
   ( Algorithm: RSA-with-SHA-1 )
   ( N: =Hi7KugV013Tv978d00vCpQ== )
   ( D: #43 ) )
( Shared-Secret-Key:
   ( Algorithm: RC5-32/12/16-CBC )
   ( K: "ossifrage" ) )
```

Public keys are for signature verification and/or encryption (depending on the algorithm), while private keys are for the corresponding operations of signing and/or decryption. A public-key algorithm used for signing or signature verification must specify a hash algorithm.

Shared secret keys can be used for encryption, decryption, or for computing MAC's (message authentication codes). A shared secret key could be a shared password, for example. A shared secret key might also be a time-varying password (as obtained, say, from a Security Dynamics SecurID card).

4.2 Principals as public keys, and their servers

This section describe the fundamental notion of a "principal," defined in terms of a public signature verification key, and talks about simple interactions with a principal.

A SDSI principal is defined as a public signature verification key, one or more optional global names, and one or more optional internet addresses. The most important thing about a principal is its ability to verify signed statements; that is why a principal is defined in terms of its public key. We say that the principal "makes" the signed statements, even if its key is only verifying signatures made by the corresponding private key.

```
( Principal:
  ( Public-Key: ... )
  ( Global-Name: ( ref: VeriSign!! WebMaster Bob-Jones ) )
  ( Principal-At: "http://abc.webmaster.com/cgi-bin/sdsi-server/" )
  ( Server-At: "http://xyz.webmaster.com/cgi-bin/sdsi-server/" )
  )
```

The optional `Global-Name:` field suggests one or more "global names" for the principal, starting from a SDSI distinguished root. These names are merely suggestions as to how one might obtain the relevant certificates if one wants to verify that the global names are indeed correct.

The optional `Principal-At:` field specifies an Internet address (typically as a partial URL) for the principal. Queries to the principal can be addressed there, although the principal's servers (if any) should normally be tried first.

The optional `Server-At:` field specifies the Internet addresses of one or more "servers" who can distribute certificates and other signed statements on behalf of the principal. If more than one server is specified, it is understood that the servers are equivalent, and that a query could be addressed to any one of them. The servers should have high reliability and high on-line availability, whereas the principal may be frequently or even permanently off-line.

A server also distributes the principal's "auto-certificate," an object signed by the principal that gives additional information about that principal. Auto-cert service is similar to the standard "finger" service, except that the reply is a SDSI auto-certificate. We also imagine that there could be standard services similar to Alta Vista that might also be able locate an auto-cert given the public key.

Information in the principal other than the public key is considered as merely "advisory"; two principals are taken as being "the same" if and only if they have the same public key.

There are two reasons for keeping the object of type `Principal:` as short as possible by pushing as much information as possible into the auto-certificate:

- Information that may change should not be included in the `Principal:`, since a change would require re-issuing any certificates about that principal. Putting such information in the auto-cert enables it to be modified without changing any certificates other than the auto-cert itself.

- The `Principal:` object normally appears in every object signed by that principal, and so it is desirable to economize. Including a large GIF photo of oneself is reasonable in an auto-cert, but not in every message one signs.

Individuals who expect to utilize their `Principal:` object for a long time should choose their servers carefully, since reliability and longevity are critical. One needn't specify one's own machine as one's server; indeed, this may be a poor choice. Organizations may provide servers for their members, or commercial services may provide servers for a fee. For short-term principals, such as many "roles", using one's own machine may be satisfactory.

4.3 Encrypted objects

This section describes how to represent an encrypted object within SDSI, as a list containing both a key indicator and the ciphertext.

The encryption of an object X is an object whose attribute/value fields indicate the encryption key (either a public key or a shared-secret key) used to encrypt the object and give the ciphertext resulting from encrypting an ASCII representation of X.

The ciphertext may be an arbitrary sequence of S-expressions; some encryption algorithms may find this convenient. For example, the IV and ciphertext can be represented as separate octet-strings.

The object type may either be `Encrypted-Macro:` or `Encrypted:`. The understanding is that any object of type `Encrypted-Macro:` that a party receives during a protocol should be immediately decrypted by that recipient, and replaced by the decrypted contents. The `Encrypted-Macro:` objects are just temporary envelopes; they recipient should open and discard them, keeping the contents instead. (The `Encrypted-Macro:` type is like a macro whose definition is the decrypted contents.) It is an error if an `Encrypted-Macro:` can not be decrypted. On the other hand, an `Encrypted:` object should only be decrypted "as necessary" during other processing. In all other respects, these two types are essentially the same.

The encryption key is indicated by

- Giving it explicitly in a `Key:` field:

  ```
  ( Encrypted:
    ( Key: ( Shared-Secret-Key: ... ) )
    ( Ciphertext: =Yh87oKlqpBv8iY55+n== ... ) )
  ```

- Giving its hash in a `Key-Hash:` field:

  ```
  ( Encrypted:
    ( Key-Hash: ( SHA-1 #241dc... ) )
    ( Ciphertext: =Yh87oKlqpBv8iY55+n== ... ) )
  ```

- Representing it explicitly as an encrypted object:

  ```
  ( Encrypted:
    ( Key: ( Encrypted:
               ( Key-Hash: ( SHA-1 #548... ) )
               ( Ciphertext:  #765... ) ) )
    ( Ciphertext: #345... ) )
  ```

  Here `#765...` denotes the encryption of the message key by an encryption key (public or shared-secret) with hash `#548...`, and `#345...` is the encryption of the message object with the message key.

- Giving a `( ref: ... )` expression that evaluates to the key.

```
( Encrypted:
  ( Key: ( ref: *SENDER key-49 ) )
  ( Ciphertext: ... ) )
```

Here the `*SENDER` macro is used.

In the last example, one way that the receiver might know what value the sender has for `key49` is that the sender might previously have sent him an `Encrypted-Macro:` certificate including the definition:

```
( Cert:
   ( Local-Name: key-49 )
   ( Value: ( Shared-Secret-Key: RC5 ... ) )
   ( Signed: ... ) )
```

4.4 Signed objects

This section describes how to represent signed objects with three signature formats (wrapped-hash, wrapped-object, and signature appendix), and shows how to transform between these formats without performing cryptographic operations. It also shows objects can be "co-signed;" with either parallel co-signatures or cascaded co-signatures.

A signed object can be signed in either "wrapped" mode (where the signed-object contains the object being signed or its hash) or in "appendix" mode (where the signature is appended to the end of the signed object). One can created cascaded co-signatures (in either mode) and parallel co-signatures (in appendix mode only).

In any case a signature is represented by an object of type `Signed:`. An example of a minimal object of type `Signed:` is shown below:

```
( Signed:
  ( Object-Hash: ( SHA-1 =7YhdOmNcGFE071QTzXsap+q/uhb= ) )
  ( Date: 1996-02-14T11:46:05.046-0500 )
  ( Signature: #3421197655f0021cdd8acb21866b ) )
```

Such an object, by itself, is said to be a signature in "wrapped-hash" format.

The mandatory `Object-Hash:` field specifies the hash of the object being signed. An arbitrary object may be signed, although this paper only talks about signed certificates or messages.

The mandatory `Date:` field gives the (effective) date of signing. There is of course nothing to prevent a signer from pre-dating or post-dating his signatures.

The mandatory `Signature:` field gives the output produced by the signature algorithm, which can be an arbitrary sequence of S-expressions, obtained by hashing the entire `Signed` object, except for any attribute/value pairs of type `Signature:` or `Object:`, and then applying the signature algorithm to the result.

We now describe the optional fields for a `Signed:` object.

```
( Object: ... )
```

Th optional `Object:` field explicitly gives the object being signed, whose hash is given in the `Object-Hash:` field. (The signature verification procedure must check this.) When this field is included, the `Signed:` object is said to be in "wrapped-object" format. This field is not hashed by the signature algorithm, since its hash is already given in the (mandatory) `Object-Hash:` field, which *is* hashed by the signature algorithm. This convention also enables transforming between different signature formats, as described below, because the `Object:` field can be removed while leaving the signature valid.

```
( Expiration-Date: 2000-01-01-00:00:00.000-0500 )
```

The optional `Expiration-Date:` field gives the date after which the signature is no longer valid (even if reconfirmed by a server).

```
( Signer: ( Principal: ... ) )
```

This optional `Signer:` field specifies the signing principal. This field must be included whenever the signed object might later be used in a context where it is not obvious who the signer is, or if the signature requires periodic reconfirmation. This field should be in the first signed message sent by each party in a protocol, and may be omitted thereafter except for messages (such as membership certificates) that will be kept around after the protocol.

```
( Signing-Key-Hash: ( SHA-1 #7613445... ) )
```

The optional `Signing-Key-Hash:` field is used when the signature (here a MAC) is created with a shared secret authentication key; it specifies the hash of the shared-secret key.

```
( Credentials: ... )
```

The optional `Credentials:` field lists various "credentials" of the signer. These take the form of certificates that may help to establish that the signer has the appropriate authorizations, such as what might be needed to access data that is restricted by an access-control list.

```
( Signing-For: ( Principal: ... ) )
```

The optional `Signing-For:` field is used when the signer is signing on behalf of someone else. The most common situation occurs when a server is signing something (such as a membership certificate) on behalf of a principal. When this field is present there must also be a `Credentials:` field present that demonstrates that the signer indeed has the authorization necessary to sign for the other party.

```
( Re-confirm: P1M ( Principal: ... ) )
```

The optional `Re-confirm:` field specifies that reconfirmation is required, and gives details. In this example, this field says that one must periodically "re-confirm" the signature every month. The period is specified according to ISO 8601:1988 format for periods without start or end. This begins with a `P`, followed by an optional time-period indicated as so many years (Y), months (M), weeks (W), days (D), optionally followed by a `T`, followed by so many hours (H), minutes (M), and/or seconds (S). Thus, `PT8H` says that reconfirmation is required every 8 hours, and `P6WT4H3M` says that reconfirmation is required after 6 weeks, 4 hours, and 3 minutes. A signature needs to be reconfirmed if its date (or the date of its last reconfirmation) is more than re-confirmation period old.

The next optional value within the `Re-confirm:` field is the "reconfirmation principal," given if different than the signing principal. (It might typically be a server.) Section 6.2 describes reconfirmation procedures.

Wrapped-hash, wrapped-object, and appendix signature formats.

An object of type `Signed:` by itself is a "wrapped" form of the signed object. It may or may not explicitly contain the object X being signed. In either case the hash of X is given. If X is also explicitly given, we say that the signature is in "wrapped-object" form. If only the hash of X is given, we say that it is in "wrapped-hash" form.

A signature in wrapped-object form can be "unwrapped" and converted into two objects: the object X itself and a `Signed:` object in wrapped-hash form that contains only the hash of X. Because the signature algorithm produces the same result whether or not the `Object:` field is present, this unwrapping operation does not require any cryptographic operations.

An object of type `Signed:` can be therefore be detached and managed separately (in wrapped-hash form) from the object itself. Many applications require this capability.

An S-expression that is a *list* admits another signature format, which we call an *appendix-mode* signature. To create such a format the `Signed:` object in wrapped-hash form is added as an appendix to the end of the list object being signed, as in:

```
( Get.Query:
   ( Template: ( Auto-Cert: ) )
   ( Signed:
     ( Object-Hash: ( SHA-1 #345678 ) )
     ( Date: 1996-02-14T11:46:05.046-0500 )
     ( Signature: #8dff4123 ) ) )
```

Verifying the signature thus requires first removing the `Signed:` field in order to compute the hash of the original list.

Appendix mode is the "standard" signature mode for SDSI objects and messages because it is the only mode that also allows co-signatures. However, there there is no appendix format for signing octet-strings, so wrapped mode is mandatory for applications that must sign octet-strings representing word-processing documents, executable binary files, etc.

Because of the way the signature algorithm is defined, one can also transform easily between a wrapped-object form of a signed *list* X and the appendix form of the signed list X. This requires only simple re-arrangement; no cryptographic operations are required. Thus

```
( Signed:
  ( Object-Hash: ( SHA-1 #345678 ) )
  ( Object: ( Get.Query: ( Template: ( Auto-Cert: ) ) ) )
  ( Date: 1996-02-14T11:46:05.046-0500 )
  ( Signature: #8dff4123 ) )
```

is equivalent to the previous example.

Co-signatures: Parallel and cascaded.

A list can have more than one signature appendix; these are "parallel co-signatures" that sign everything up to but not including the previous signature(s). Parallel co-signatures are the result of two or more signers independently signing an object, and appending their independent signatures to the end of the object.

```
( Get.Query:
   ( Template: ( Auto-Cert: ) )
   ( Signed:
     ( Object-Hash: ( SHA-1 #345678 ) )
     ( Date: 1996-02-14T11:46:05.046-0500 )
     ( Signer: ( Principal: ( Global-Name: VeriSign!!'s "Bob" ) ... ) )
     ( Signature: #8dff4123 ) )
   ( Signed:
     ( Object-Hash: ( SHA-1 #345678 ) )
     ( Date: 1996-03-19T07:00:11.341-0500 )
     ( Signer: ( Principal: ( Global-Name: VeriSign!!'s "Alice" ) ... ) )
     ( Signature: #a78300b3 ) ) )
```

Another form of multiple signature is the "cascaded co-signature" obtained by adding a signature to the end of a `Signed:` object. Cascaded co-signatures are cumulative; they effectively sign all previous material. A cascaded co-signature is what you get if you sign the `Signed:` object at the end of a previously signed list.

```
( Get.Query:
   ( Template: ( Auto-Cert: ) )
   ( Signed:
     ( Object-Hash: ( SHA-1 #345678 ) )
     ( Date: 1996-02-14T11:46:05.046-0500 )
     ( Signer: ( Principal: ( Global-Name: VeriSign!!'s "Bob" ) ... ) )
     ( Signature: #8dff4123 )
     ( Signed:
       ( Object-Hash: ( SHA-1 #86731b ) )
       ( Date: 1996-03-19T07:00:11.341-0500 )
       ( Signer: ( Principal: ( Global-Name: VeriSign!!'s "Alice" ) ... ) )
       ( Signature: #a78300b3 ) ) )
```

Because the second `Signed:` object contains the hash (`( SHA-1 #86731b )` of the first `Signed:` object, and the first `Signed:` object contains the hash (`( SHA-1 #345678 )` of the enclosing S-expression, Alice's second signature effectively signs both the original S-expression `( Get.Query: ... )` and Bob's first signature. One can have cascaded co-signatures in either wrapped or appendix mode.

An application of cascaded co-signatures is digital time-stamping, where the second signature provides evidence that the first signature had already been created at the time the second signature was applied.

Another use would be where an application program running on behalf of a principal signs the signature created by the principal, so that the server knows that the request not only came from an authorized principal, but from an authorized principal running the correct program.

5 Local Names, Certificates, and Name/Value Bindings

This section describes how each principal may define his own local name space, and how he may issue certificates to export his local name/value bindings.

5.1 Names

Each principal has its own local name-space.

A name may be represented in one of two ways:

- As an octet string that does not begin with any special character. Examples: `"abc"`, `mary-sue`, `tom@nsf.gov`, `#61` .
- As an arbitrary S-expression `n`, enclosed in the form `( Local-Name: n )`. Example:

```
( Local-Name: ( Accounting ( Bob Smith ) ) )
```

An octet-string name `n` can be thought of as equivalent to the form `( Local-Name: n )`; they evaluate to the same thing. We expect that most names will be represented as octet-strings written as tokens.

S-expressions of the forms:

```
( Principal: ... )
( Group: ... )
( Quote: ... )
```

may not be used as local names; these expressions are always treated as constants (they evaluate to themselves).

Although SDSI allows arbitary S-expressions as names, we suspect this feature will not be used much. However, one reason we have included this capability in SDSI is that its flexibility provides a path for attempting to accommodate X.509 distinguished names within SDSI.

The principal can choose names arbitrarily. He may also partition his name space with multi-part tokens (`group:joggers`), hierarchical tokens (`fs:/sys/bin/foo.exe`), or lists.

We call names in SDSI "local-names" to emphasize the fact that they are in a name space local to some principal, rather than in some global name space.

5.2 Name/Value Bindings

A local name can be undefined, or it can be bound to some value (i.e., some object). The principal may assign a value to a local name by issuing a corresponding certificate. If the local name already has a valid name/value certificate, the new certificate augments the old one, in the sense that a SDSI application is deemed to act correctly if it uses the name/value binding given in either certificate.

The binding can be "symbolic"; `bob` can bind his local name `lawyer` to `ted's lawyer`.

Recall that DNS names are treated in a special manner. The mapping from DNS names to `ref:` expressions works as follows, when attempting to interpret any octet-string:

- If the octet-string has locally defined value, then that value is taken as the desired value. A principal can, for example, define "`bob@fudgeco.com`" to have the desired value directly.
- Otherwise, if the octet string is of the form `name@ak...a2.a1` for some $k \geq 1$, then the string is interpreted as equivalent to the first form of the following list that has its first component defined in the local name space:

```
( ref: ak...a2.a1.DNS!! name )
( ref: a(k-1)...a2.a1.DNS!! ak name )
( ref: a(k-2)...a2.a1.DNS!! a(k-1) ak name )
...
( ref: a1.DNS!! a2 ... ak name )
( ref: DNS!! a1 a2 ... ak name )
```

This mechanism allows one to write ordinary email addresses in an ACL, and get the correct interpretation, leveraging off the DNS security hierarchy (which is soon to appear, we presume). The name `DNS!!` is a SDSI "distinguished root," and may not be re-defined. The other names ending in "`.DNS!!`" may be redefined by the user to eliminate dependence on the DNS servers, as a security enhancement. (For example, Microsoft users can have `microsoft.com.DNS!!` defined locally, to ensure that addresses ending in `microsoft.com` are always resolved correctly, even if the DNS servers are corrupted.

Details of `ref:`

Here is some detail about how `ref:` works:

```
      ( 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.
```

5.3 Certificates

Certificates (certs) are signed objects or messages. The body of these signed objects or messages consists of a list of attribute/value pairs, followed by the signature(s).

Most certificates contain a `Local-Name:` field giving a local name. The certificate is "about" that local name. The name may be an arbitrary S-expression.

(As a minor point, we note that if a certificate is signed with a cascaded co-signature (as you can get with a re-confirmation) then the definition is understood to be only for the primary signer's name space, not for the secondary principal. On the other hand, if the certificate has parallel co-signatures, then the certificate is effective for each signer, since all but any one of the co-signatures can be removed, still leaving a validly signed certificate.)

The certificate may contain a `Value:` field. If so, then this certificate serves to bind the given local name to that value; it is a "name/value certificate." The value can be any SDSI object; typically it is a principal or a group definition. The value should be one of the following:

- a `Principal:` object,
- a `Group:` object,
- a `Quote:` object.
- a `ref:` object,
- an `Encrypted:` object, that, when decrypted, gives one of the allowed types in this list,
- an `Assert-Hash:` object whose first field is an object of one of the allowed types in this list.

Fetching the value of the `Value:` field basically just takes the value as stored, except that encrypted objects are decrypted and Assert-Hash wrappers are removed, until an object of one of the first three types is obtained. It is an error if an `Encrypted:` object can not be decrypted. `Assert-Hash:` checks are performed as the recursion unwinds, and it is an error if an `Assert-Hash:` check fails. It is an error if an object is obtained that is not one of the above types. (Fetching the value of other fields is similar in that `Encrypted:` and `Assert-Hash:` wrappers are removed whenever possible; these wrappers are stored with the certificate, but removed when these values are accessed.)

The certificate may also contain a `Description:` field intended for human consumption. If so, then this certificate can also be viewed as an "identity certificate." Understanding the `Description:` field is not required for accepting a local name/value binding, but this field should be read and understood if the certificate is used for any other purpose.

The intended typical use of the description item is to describe the entity controlling the specified principal, and to give any additional information or policies used in verifying his identity, so that a human can decide whether or not to place the principal into his data-base under some local name (possibly a different name than that in the certificate).

The certificate may contain arbitrarily many other fields with corresponding attribute values.

Here is a certificate defining the local name "`Q`" as a principal, and also acting as an identity certificate, since it has both a `Value:` field and a `Description:` field.

```
( Cert:
   ( Local-Name: Q )
   ( Value: ( Principal: ... ) )
   ( Description:
     [text/richtext]
     "Bob Q. Smith has worked at <bold>WebMaster International</bold>
     as a link repairman since 3/4/95.  Bob has blond hair, is
```

```
    25 years old, and has <italic>green</italic> eyes." )
  ( Phone: 617-665-0923 )
  ( Signed: ... ) )
```

Here is a group definition for "`FudgeCo-employees`":

```
( Cert:
   ( Local-Name: FudgeCo-employees )
   ( Value: ( Group: "Bill Sweet" "Candy Tooth" "Ima Nut" ) )
   ( Description:
           "All current hourly and exempt employees including
            those on medical or parental leave." )
   ( ACL: ( read: FudgeCo-management ) )
   ( Signed: ... ) )
```

Here is a symbolic definition of "`head-nurse`". (Note that it presents the `Value:` field with a bit of syntactic sugar...)

```
( Cert:
  ( Local-Name: head-nurse )
  ( Value: Mass-General's head-nurse )
  ( Description: "The current head nurse at Mass General Hospital." )
  ( Signed: ... ) )
```

The "type" of a certificate need not be `Cert:`; it is possible to have various types of certificates.

Note that I can issue one certificate with no expiration date but with monthly reconfirmation required. A new certificate for the same local name can be issued to replace it, and the reconfirmation procedure can cause appropriate queries to be made to get everyone up to date within one confirmation period.

It is intended that there would typically be only one valid certificate by a signer about a particular local name at a time. However, if there is more than one valid certificate in existence, then it is perfectly valid for a server to use *any* one of the available valid certificates. In particular, it is expected that servers may have a policy of consistently using the most recent one available (i.e., the one with the most recent signing date), wherever possible. Thus a credential submitted by a client may be disregarded in favor of a more recent certificate known to the server. This holds as well for "computed" certificates such as member certificates. (In this case, the certificate is "about" the principal and the group collectively, so it can only be replaced by another more recent certificate about the same principal and group.) By having his servers implement such policies, and by having efficient distribution mechanisms to his servers, a principal can be reasonably effective about revising the binding given to a local name, even before the actual expiration date of the earlier definition. This is a pragmatic point of view; technically the older certificates are still valid until they expire.

6 Protocols

This section describes the SDSI framework for two-pary communication protocols. It discusses the standard "Get" protocol for obtaining information from a server, the protocol for reconfirming a signature, and the protocol for obtaining an auto-certificate. It also discusses delegation certificates and transport mechanisms.

Communication in SDSI takes place as a sequence of *protocols* between two parties. Typically one party is called the "client" or "requestor," and the other party is called the "server."

In a protocol the client typically initiates the protocol by sending a first message to the server, and then the server and client alternately send messages to each other until the protocol is finished. However, a party may send two or more messages in a row to the other party without waiting for a reply, and the parties may even send messages to each other simultaneously, if the protocol allows it.

We make no assumptions about how the messages in a protocol are transported, nor about the confidentiality, authenticity, integrity, or reliability of the transport mechanism. Some discussion of transport mechanisms is given in Section 6.5.

Each protocol defines standard message types and patterns of interaction. A typical protocol might be a query/response protocol wherein the client asks a server for some information. Although we only define a few protocols in this paper, a SDSI design goal is to provide a convenient framework for incorporating other protocols.

When a message is transmitted, it may be sent in compressed form, using the compression techniques of Section 3.3. When it is received, it is immediately decompressed before further processing.

After decompression, the recipient decrypts the message (or any portions of it) as appropriate. More precisely, the recipient will immediately decrypt any objects marked as `Encrypted-Macro:`; other encrypted objects are not decrypted until necessary.

Many messages are signed. If a signature fails to verify, the recipient sends back an error message.

6.1 Queries with the "Get" protocol

A server holds a database of certificates, and can be queried to return collections of certificates that satisfy some criteria. The SDSI protocol for this is called `Get`.

The `Get` query always contains a `To:` field specifying a principal. The certs returned must have that principal as a primary signer.

A query specifies a "template" for the desired certificates, giving the object type of the desired certificates (such as `Cert:` or `Auto-Cert:` or ...), and a list of attribute/value pairs that the desired certificates must contain. For example, the search criteria

```
( Template: ( Cert: ( Local-Name: bob ) ) )
```

is a request for all objects of type `Cert:` that contain an attribute/value pair that is equal to `( Local-Name: bob )`. (In testing for equality, no evaluation of names is done, so that one is comparing to see if the local name is literally the octet-string `bob`.) A search criteria of the form:

```
( Template: ( Cert: ( Value: '#23 ) ( Color: Red ) ) )
```

asks for all certs with the given `Value:` and `Color:` fields.

More generally, the `( Template:` *form*) field may have an arbitrary S-expression *form* as its value. That is, the *form* is a template; a matching object must contain components matching with each component of the specified template. More precisely: the form matches an object if and only if:

- the form and the object are equal octet-strings, or

- the form and the object are both nonempty lists, and
  - the first element of the form matches the first element of the object, and
  - each element of the form other than the first matches some element of the object other than the first.

The server replies with a list of certs satisfying the query or with an error message.

Here are some sample queries. The first asks for the current certificate about `jim`:

```
( Get.Query:
  ( To: ( Principal: ... ) )
  ( Template: ( Cert: ( Local-Name: jim ) ) )
  ( Signed: ... ) )
```

The second asks for all certificates that have a specified principal as the given value.

```
( Get.Query:
  ( To: ( Principal: ... ) )
  ( Template: ( Cert: ( Value: ( Principal: ... ) ) ) )
  ( Signed: ... ) )
```

The query may also contain the optional

(`Encrypt-Reply:` *flag key-indicator*)

field if the reply may have to be encrypted. The *flag* is either `mandatory` or `optional`. If `mandatory`, the client insists that the reply be encrypted with that key. If `optional`, the reply need only be encrypted if the server finds it necessary (say if the reply includes material whose distribution is restricted by an ACL). If the `Encrypt-Reply:` field is absent, the server may return an error, or use an encryption key from the *client*'s auto-certificate. After the flag comes the encryption *key-indicator*, which may be a public encryption key, or the hash of a public encryption key, or an encrypted key.

Similarly, the query may contain an optional `Sign-Reply:` field indicating whether or not the reply should be signed:

(`Sign-Reply:` *flag*)

The *flag* field can be `mandatory` or `optional`. A missing `Sign-Reply:` field is equivalent to a `mandatory` flag. Some protocols may dictate that the reply be signed, in which case the server has no choice but to sign. In other protocols, the reply is only optionally signed, and the server can be forced to sign his reply by having a `Sign-Reply:` with `mandatory` flag in the query. The server generally signs error replies, but need not sign replies with content that is already signed (e.g. a certificate), unless the client demands it. The server may refuse to honor a request for a signature if it is too busy, but in this case the server's auto-certificate must state that it might do so. If the server gives a signature, it will typically include a `Credentials:` field in its reply, giving the `Delegation-Cert:` authorizing it to act as a server for the principal.

The reply to a `Get.Query:` has the following format:

```
( Get.Reply:
  ( Your-Last-Message-Hash: ( SHA-1 =tGbiO+dc...= )
```

```
( Reply:
   ( Cert: ... )
   ( Cert: ... )
   ... )
( Signed: ... ) )
```

The `Your-Last-Message-Hash:` field gives the hash of the query being responded to (including all of its signatures). This should uniquely identify the query, assuming that no two queries are identical (due to their having different signing times, for example).

The reply may contain more than one cert. The reply may be signed to prevent an adversary from deleting some of the certs. Various error messages can be generated, such as might happen if the reply exceeds some pre-specified size.

A reply or error message does not need to repeat the addressee or the query, since it is included implicitly in the `Your-Last-Message-Hash:` field.

```
( Get.Error:
  ( Your-Last-Message-Hash: ( SHA-1 =tGbiO+dc...= )
  ( Error: ... )
  ( Signed: ... ) )
```

The error message may be signed by the principal's server, instead of by the principal queried.

6.2 Reconfirmation Queries

SDSI does not have "certificate-revocation lists" as a means of revoking the signature on a previously signed object. Instead, signatures may be designated as needing periodic reconfirmation. The signer can specify the reconfirmation period that is appropriate for that signature; some signatures might only need to be re-confirmed yearly, while others might need reconfirmation hourly.

A signature with reconfirmation specified is considered valid if it has been reconfirmed by either the original signer or the reconfirmation principal within the most recent reconfirmation period, and the reconfirmation signature has not expired.

A reconfirmation query takes the form:

```
( Reconfirm.Query:
  ( To: ( Principal: ... ) )
  ( Signed-Object:
     ( Signed:
        ( Object-Hash: ( SHA-1 #5128 ) )
        ( Date: 1999-12-25-08:00.000-0500 )
        ( Signature: #333111 ) ) )
```

The `Signed-Object:` field gives the wrapped-hash form of the signed object whose signature is to be reconfirmed.

A successful reconfirmation is either:

- A new wrapped-hash signature of the original object by the original signer, or
- A new wrapped-hash signature that is a cascaded signature for the wrapped-hash signature of the original object, signed by the reconfirmation principal. (The added signature does not itself contain any `Reconfirm:` field, although it may contain an expiration date.)

```
( Reconfirm.Reply:
  ( Your-Last-Message-Hash: ( SHA-1 =Ac8wE1...= ) )
  ( Signed-Object:
     ( Signed:
        ( Object-Hash: ( SHA-1 #5128 ) )
        ( Date: 1999-12-25-08:00.000-0500 )
        ( Signature: #333111 )
        ( Signed:
           ( Object-Hash: ( SHA-1 #a783b0 ) )
           ( Date: 2000-01-25-12:10.000-0500 )
           ( Signature: #86723 ) ) )
```

The `Your-Last-Message-Hash:` field gives the hash of the reconfirmation query (including all of its signatures). No signature is necessary on the message itself, since the object itself has been effectively (re-)signed.

If the signed object is a certificate, and that certificate is still valid, but has been superseded by another certificate with the same name, then the *new* certificate is returned with a signature.

A failure of reconfirmation takes the form

```
( Reconfirm.Reply:
  ( Your-Last-Message-Hash: ( SHA-1 =Ac8wE1... ) )
  ( Failure: reason )
  ( Signed: ... ) )
```

The failure reply is signed by either the reconfirmation principal or the original signing principal.

6.3 Auto-Certs

An auto-certificate is a special kind of certificate. It is distinguished by having been signed by the principal whom it is about, and by having the object type `Auto-Cert:`.

Each principal must have an auto-certificate. The auto-cert may give additional information not given in the `Principal:` object. This auto-certificate is signed by the principal itself. This information may expire or be changed without changing the `Principal:` object. The `Principal:` object specifies one or more auto-cert servers that can respond to a query with the auto-cert.

Since this information is signed by the principal itself, it has no third party vouching for it, and should not be trusted without suitable corroboration.

```
( Auto-Cert:
  ( Public-Key: ... )
  ( Principal-At: ... )
  ( Server: ... )
  ( Local-Name: ... )
  ( Global-Name: VeriSign!!'s Wonderland's "Alice McNamee")
  ( Name: [charset=unicode-1-1] #764511fcc... )
  ( Description: ... )
  ( Encryption-Key: ( Public-Key: ... ) )
  ( Postal-Address: ... )
  ( Phone: ... )
```

```
    ( Fax: ... )
    ( Photo: [image/gif] =Yu7gj9D+zX2C... )
    ( VeriSign-Cert: [application/X.509v3] =GvC492Sq... )
    ( Email-address: AliceMcNamee@wonderland.com )
    ( Signed: ... ) )
```

The `Public-Key:`, `Principal-At:`, and `Global-Name:` fields duplicate the corresponding fields (or define the latter fields if they are missing) in the `Principal:` whom this auto-certificate is about.

None of the fields, except the `Public-Key:` and `Signed:` field, are required. The only fields that are intended to be machine-readable are the `Principal-At:`, `Server:`, `Email:`, `Global-Name:`, and `Signed:` fields. The other fields are there primarily for human consumption, and can be added at one's discretion, following whatever standard coding conventions exist.

A `Server:` field must be supplied if the principal wishes to export any bindings, or group information. A `Server:` field specifies one or more "servers" for the principal: an entity that can respond to arbitrary queries on behalf of that principal. A server for a principal can be viewed primarily as an agent with a database of statements signed by that principal. The server will produce these statements upon request, assuming that the client has adequate permission. A principal may specify itself as a server. The principal may designate more than one server; they are equivalent, and a client may address a query to any one of them.

The separation of the functions of "principal" and "server" affords flexible design permitting high availability of signed statements while offering the opportunity of keeping the principal's private keys off the server machines.

In many cases, a server is merely distributing statements previously signed by the principal. However, error messages will always be signed by the server.

The `Local-Name:` field gives the principal's favorite nickname for himself. One can consider using this nickname in one's own space for this principal, but it is not required.

The `Description:` field gives arbitrary free-form text by the entity controlling the key about himself.

The optional `Email-Name:` field gives a list of one or more names by which this principal can be called, starting from one of the distinguished SDSI roots. (In the example shown, the implicit root is `DNS!!`, by the special rules for handling such names.) This address can also be used for sending email to the principal.

The standard `Get` protocol can be used to request auto-certificates:

```
( Get.Query:
  ( To: ( Principal: ... ) )
  ( Template: ( Auto-Cert: ) )
  ( Signed: ... ) )
```

6.4 Delegation Certificates

The `Delegation-Cert:` is used to authorize a group (of servers) for to be able to sign on behalf of the signing principal. `Delegation-Cert:`'s may be used as credentials by the server, to show that their signatures are authorized by the principal.

```
( Delegation-Cert:
  ( Template: form )
```

```
   ( Group: group )
   ( Signed: ... ) )
```

This certificate authorizes every member of *group* to be able to sign objects matching *form* on behalf of the signing principal. For example, the following certificate:

```
( Delegation-Cert:
  ( Template: ( Membership.Cert: ( Group: fudge-lovers ) ) )
  ( Group: ( Principal: ... (A) ... ) )
  ( Signed: ... ) )
```

authorizes the specified principal `(A)` to be able to sign membership certificates for the group `fudge-lovers`.

The matching routine is the same as for the `Template:` field in a `Get.Query:` object.

Although `Delegation-Certs:` are intended primarily for principals to delegate signature authority to their servers for specific purposes, it is easy to imagine other possible uses for them, as well.

6.5 Transport mechanisms

We envision that a separate SDSI transport protocol can be designed on top of TCP/IP. Ultimately, this would be the preferred solution.

In the meantime, SDSI can also take advantage of other protocols, such as email, http, or gopher, for transport. For example, a `Get.Query:` object can be given as the content of an email message addressed to a SDSI server. Or, one could have an http URL of the form:

```
http://server.address.edu/cgi-bin/sdsi-server?GQ
```

where `GQ` is an ASCII representation of the `Get-Query:` object (with appropriate re-encodings of special characters, as usual). The replies can be given in a corresponding manner.

We omit further suggested details here.

7 Groups and ACLs

This section describes SDSI groups, shows how groups can be defined, how membership in groups can be determined, how one can query a server to get a membership certificate, and how one can create ACL's that give permissions to members of various groups.

Intuitively, one can think of a group as just a set of principals. It is worthwhile defining groups for many security purposes, since this gives a convenient local name for the set of principals allowed to access some data or to perform some other action.

A group does not have an associated public key; there is no way for the group to make statements as such. No principal is authorized to speak for the group, although the owner of the group can change its definition. A member of the group can, however, offer his group membership certificate as a credential when he makes a query or request.

The simplest group is just the singleton set containing just one principal. An object of type `Principal`, or a local name evaluating to such an object, suffices to denote such a group.

Another simple group is the group containing everyone. This group is denoted by the reserved word `ALL!` . (This local name always has the definition `( AND: )` , which is always satisfied.)

Groups can be defined by listing their members. For example:

```
( Group: tom mary bill george ( Principal: ... ) )
```

defines a group of five members, four given indirectly (by their local names), and one directly (as a principal).

Groups can also be defined recursively in terms of other groups, using standard set-theoretic operations:

```
( Group: ( AND: friends over-18 jocks ) )  -- intersection
( Group: ( OR:  faculty staff friends ) )  -- union
( Group: ( NOT: staff ) )                  -- ALL! - group
( Group: ( MINUS: staff friends )          -- staff that are not friends
( Group: ( ANY: 2 vp-finance tom joe ) )   -- threshold of >= 2
( Group: ( OR:
           "Mary Smith"
           friends
           mit's faculty
           ( ref: #32 )
           ( Principal: ... ) ) ) )
( Group: ( OR: alpha ( AND: beta gamma ) ( NOT: delta ) ) )
```

Groups can be defined with nested expressions, as in the last example. To be a member of this group, one must be a member of `alpha`, or be a simultaneous member of `beta` and `gamma`, or not be a member of `delta` .

Note that the notation

```
( Group: a b c d )
```

has exactly the same meaning as:

```
( Group: ( OR: a b c d ) ) .
```

The use of `NOT:` can cause confusion, as can `MINUS:`. First, we note that `( MINUS: A B )` is treated as being fully equivalen to `( AND: A ( NOT: B ) )`. Second, we note that to establish a negative (such as `( NOT: B )`, which means that a principal is not a member of group `B`), it does not suffice merely to fail to find a certificate asserting that the principal is in group `B`. Rather, the owner of the group definition must have provided a certificate saying that the principal is NOT in group `B`.

We note that groups aren't actually restricted to being sets of principals, but can in fact be sets of arbitrary S-expressions. For example, a bank can have a group called `expired-accounts` whose members are objects representing the expired accounts:

```
( Group: '( Account: 3451-223-5624 )
         '( Account: 7621-004-6722 )
         '( Account: 9821-868-4110 ) )
```

(The quoting is necessary.) As another example, a company might have a set of approved software programs, the group `approved-software` might contain the hashes of the approved programs. The notation and procedures for dealing with sets of principals generalize smoothly to handle these extensions.

7.1 An algorithm for determining group membership

This section describes how it can be determined whether some individual x is a member of some group g.

A server maintaining a group definition may need to determine whether a given principal x is, or is not, a member of some group g. This may happen, for example, because x has requested information from the server, and it is necessary to determine if x is authorized. Or, the server may be responding to a membership query from x himself. This section describes how the server can determine whether or not x is a member of g.

The server will make this determination with whatever local information it has (e.g. the group definition) and with whatever credentials x has supplied (e.g. other membership certificates). The server does *not* make inquiries of other servers in order to make the determination; it may simply "fail" if there is inadequate information available locally. The server is like a busy bureaucrat who says to x, "I can't tell if you are authorized or not, and I'm not going to do the research myself to make this determination. You may try again later; it may help if you can demonstrate membership in my group `mit's faculty`." Note that a "fail" response may give x some information as to what sort of credentials are needed. If x wishes to try again it is up to him to obtain additional credentials, and then to repeat his request including the credentials (e.g. a certificate from the server's principal `mit` stating that the client is a member of its `faculty` group).

We now describe a procedure for determining whether or not a given S-expression x is a member of some group g, by recursively tracing out the group definition.

MEMBER(x, g)

- If this call is a recursive invocation of an earlier, still-active call with the same arguments x and g, then return FAIL.
- If g is of the form `( Principal: ... )`, then return TRUE if $x = g$; otherwise return FALSE. (For comparing principals, only compare the `Public-Key:` fields.)
- If g is of the form `( Quote: y )`, return TRUE if $x = g$. Otherwise return FALSE.
- If g is of the form `( Group: g1 g2 ... gk )`, then return MEMBER(x,`( OR: g1 g2 ... gk )`). (If $k = 1$, this is equivalent to returning MEMBER(x,`g1`).)
- If g is of the form `( OR: S1 S2 ... Sk )`, evaluate MEMBER(x, S_i) for $i = 1, 2, \ldots, k$. If any call returns TRUE, then return TRUE. If $k = 0$ or all calls return FALSE, then return FALSE. Otherwise return FAIL.
- If g is of the form `( AND: S1 S2 ... Sk )`, evaluate MEMBER(x, S_i) for $i = 1, 2, \ldots, k$. If any calls return FALSE, then return FALSE. If $k = 0$ or all calls return TRUE, then return TRUE. Otherwise return FAIL.
- If g is of the form `( NOT: S )`, evaluate MEMBER(x, S). If this call returns TRUE, return FALSE. If this call returns FALSE, return TRUE. Otherwise return FAIL.
- If g is of the form `( ANY: #d S1 S2 ... Sk )`, evaluate MEMBER(x, S_i) for $i = 1, 2, \ldots, k$. If at least d calls return TRUE, then return TRUE. If at least $k - d + 1$ calls return FALSE then return FALSE. Otherwise return FAIL.
- If the client has supplied a set of valid credentials sufficient for establishing membership (resp. non-membership) in g, return TRUE (resp. FALSE).

- If g is of the form `( ref: ... )`, then determine the value V of the form g, and then return MEMBER(x, V). (This process may utilize credentials given by the client.) Return FAIL if V can not be determined.
- If g is an octet string, or a list of the form `( Local-Name: y )`, then attempt to determine `( ref: ` g ` )`. (Note that this process may utilize credentials supplied by the client.) If this fails, return FAIL. If this returns a value V, then return MEMBER(x, V).
- Otherwise return FAIL.

This algorithm thus detects circular definitions to prevent infinite looping. If `friends` is defined as

```
( Group: associates terry )
```

and `associates` is defined as

```
( Group: friends pat ) ,
```

then `pat` is successfully determined to be a member of `friends`, without looping.

A principal is a member of the group defined by the threshold definition

```
( ANY: 2 doctors lawyers bankers )
```

if it is a member of two or more of the groups `doctors`, `lawyers`, `bankers`.

Multiply-signed requests.

In some applications, a request will be signed by two or more clients. (Here we do not distinguish between cascaded and parallel co-signatures.) In this case, the same algorithm can be applied, where it is understood that a test succeeds anywhere during the evaluation process if any one of the signers passes the test. A request signed by `bob` and `alice` will pass the test of membership for the group

```
( AND: doctors lawyers )
```

if `bob` is a laywer and `alice` is a doctor, for example. There are other ways in which they could pass this test; for example, Bob might be both a doctor and a lawyer, and Alice might be neither.

With a multiply-signed request, the effect is that the signing principals have merged, and their respective group memberships are combined, so that they can act with more power joined together than they can individually.

7.2 Membership queries

Membership queries are used to obtain membership certificates; an individual can query a server to ask whether he is a member of a particular group; the server can respond with a membership certificate.

For very large groups, it may be too expensive to return the entire group definition to a client. In such cases, group definitions can be queried with the `Membership` protocol, which the client initiates with a `Membership.Query`:

```
( Membership.Query:
  ( To: ( Principal: ... A ...) )
```

```
    ( Member: ( Principal: ... B ... ) ... )
    ( Group: fudge-lovers )
    ( Credentials: ... )
    ( Signed: ... ) )
```

The `Member:` list specifies the principals or objects about which membership is to be determined. It typically gives just one principal, although it may contain more. (If so, the interpretation is as for the "multi-signer" interpretation given earlier.)

The `Member:` list may contain arbitrary S-expressions instead of just principals, since groups may contain arbitrary S-expressions as members. These expressions are normally quoted:

```
    ( Member: '( Account: 3451-223-5624 ) )
```

A successful reply has the form of a membership certificate:

```
  ( Membership.Cert:
    ( Member: ( Principal: ... B ...) ... )
    ( Group: fudge-lovers )
    ( Reply: answer hint )
    ( Signed: ... ) )
```

Here the principal(s) about whom the query was for are repeated in the reply, so that this reply can be passed around as a "membership certificate." The `Reply:` field gives the *answer*: one of `true`, `false`, or `fail`, with an optional *hint* in the last (`fail`) case explaining to the client how he can supply additional credentials that would eliminate the failure.

The hint is given only when the server could not completely evaluate the group definition, and so could not completely determine whether the correct answer was `true` or `false`. The hint is a "partially evaluated" group definition. That is, it is is a simplified definition of the group that is equivalent to the original definition, given what the server already knew and what credentials it had to work with. For example, it may be just an edited or simplified top-level definition of the group, with all individual principals deleted, non-failing group names or sub-forms removed, and thresholds decreased by the number of successful top-level forms contained in the original threshold form before editing. For example, if the original group definition was

```
  ( ANY: 3 A B C D E F )
```

and the client presents credentials showing he is a member of `A` and `C`, and not a member of `D`, and the server fails to otherwise determine whether he is a member of groups `B`, `E`, or `F`, then the returned *hint* could be:

```
  ( ANY: 1 B E F )
```

Similarly, if the group definition is

```
  ( OR: ( AND: A B ) ( AND: C D ) ( AND: E F ) )
```

under the same conditions, the returned hint could be

```
  ( OR: B ( AND: E F ) )
```

Given such a hint, the client can attempt to obtain sufficient certificates to make the hint evaluate to `true`, which would suffice to make him a member of the group. Of course, the client should understand that all local names in the hint are in the server's name space, not his own. To convert the definition to one that is useful to him, he would replace `B` by something like `X's B`, and so on, where the server is acting on behalf of principal `X`. To find out more about the meaning of this hint, it may be necessary to query the server for definitions of the various names it contains.

If the group definition is protected by an ACL, then the server tries to determine whether the client satisfies the ACL condition, and whether a encrypt-reply is available to encrypt the reply with. If so, the server can return a hint. If not, then no hint is given.

7.3 Access-Control Lists

The design of SDSI was motivated in part by the desire to make sure that ACL's can be written in a clear and easily auditable manner, using names for principals or groups. This section describes how that can be done.

Any (list) object may contain an ACL. For example, a group definition have an ACL so that only certain principals may read the definition. The ACL must be a top-level element of the list, and there must be at most one ACL in a list. If an ACL is present, then that list can only be accessed by principals that satisfy the ACL. SDSI ACLs can also be associated with other objects (e.g. web pages) for access control.

An ACL is of the form `( ACL: ( type1 def1 ) ( type2 def2 ) ... )` where each `type` is determines the set of operations being controlled (e.g. `read`) and where `def` is either the local name of a group, or an explicit group definition (such as a list of principals, or an algebraic definition).

As an example, the certificate for `group-23` can only be read by its members:

```
( Cert:
  ( Local-Name: group-23 )
  ( Value: ( Group: friends colleagues ) )
  ( ACL: ( read: group-23 ) )
  ( Signed: ... ) )
```

Including the ACL in the list may seem to imply that having read-access to a restricted object implies read-access to its ACL, but this can be changing the ACL to refer to a different group:

```
( ACL: ( read: group-23-readers ) )
```

where `group-23-readers` has a definition as a group, where the definition is itself protected by a different ACL:

```
( Cert:
   ( Name: group-23-readers )
   ( Value: ( Group: group-23 ) )
   ( ACL: group-23-honcho sam )
   ( Signed: ... ) )
```

Here only `sam` and `group-23-honcho` can access the definition of `group-23-readers`. Presumably, if `mary`, a member of `group-23`, requests access to `group-23`, then mary can access its definition, but not its ACL (i.e. not the definition of of `group-23-readers`).

ACLs are relevant for `Get` and `Membership` replies.

ACLs for Get replies. The default ACL for an object (in a `Get` reply) is `( ACL: )`, which allows no access. Each object must have a non-trivial ACL specified if it is to be exported outside of the local domain. An ACL of the form `( ACL: ( read: ALL! ) )` makes the object exportable to anyone who asks.

ACLs for Member replies. The default ACL for a `Membership.Cert` is `( ACL: ( member: ALL! ) )`, so that a membership query is typically answered, even if the group definition is not accessible to the client(s).

If an object has a non-trivial ACL, then the client must have supplied a encrypt-reply (or have a suitable key supplied in his auto-cert) so that the object can be returned in encrypted form.

8 Application Scenarios

This section sketches various possible applications of SDSI. Many details are omitted, but the kernel idea for each application is given.

8.1 Mail Reader

When a mail reader receives a SDSI-signed object, it can display to the user a local nick-name for the signer, if one exists in the local data-base. This nick-name can be displayed in a "secure" pop-up window that can't be spoofed by an applet or other program.

If there is no local name for the signer, then one of the signer's global names (from his auto-certificate or principal) might be displayed. Such a global name should of course be checked before displaying, and the signer might include appropriate credentials to allow such a name to be verified. Another option would be just to display the hash of the principal in such cases.

8.2 World-Wide Web

An HTML document can be extended to include an ACL, and the http protocol extended to handle a "get" with a dialogue that mimics the SDSI `Get` protocol. (Basically, you are doing a lookup on the URL rather than on the SDSI name.) For example, a document might include in its header:

```
<META NAME="SDSI-ACL" CONTENT="( ACL: ( read: mit's employees ) )">
```

using the standard `META` format for including information in the header of an HTML document.

8.3 Kerberos-like tickets

A short-lived certificate stating that x is a member of a group g can be used as a Kerberos-like ticket; it denotes authorization to use some resource that has an ACL containing g.

8.4 Distributing signed code

Code can be signed using the two-part detached wrapped-hash form. This means that the executable code itself is not modified in any way.

8.5 Corporate database access

Each record in the corporate database can be tagged with the name of the group of principals allowed to access it. In some cases, a co-signature may be needed by the application program itself. (Following the discussion of Clark and Wilson [2].)

8.6 Access to medical records

Similar to the corporate database example, except that here the hospital has the records, but the individual should perhaps be the one to authorize access. One might authorize access for a group of the form:

```
( OR: Dr.-Ruben St-Marys-physicians emergency-access )
```

where `emergency-access` is a group that is held by a trusted party, who can add a new member to it in an emergency.

8.7 Shared-secret key establishment

The techniques of this paper can be modified to handle other cryptographic protocols, such as shared-secret key establishment. This is relatively straightforward.

8.8 Multi-Cast

If I want to broadcast a confidential message to my group `banana-lovers`, I can proceed as follows. I can post a message to the `banana-lovers` Usenet news-group (or otherwise distribute it to all members) of the form:

```
( Encrypted:
   ( Key: ( ref: server's banana-recipe-241 )
   ( Ciphertext: ... ) )
```

so that all members obtain the raw ciphertext and know to ask my server for the key under the name `banana-recipe-241`. I give my server a cert:

```
( Cert:
   ( Local-Name: banana-recipe-241 )
   ( Value: ( Shared-Secret-Key: ( Algorithm: RC5-32/12/16-CBC ) ... ) )
   ( ACL: ( read: banana-lovers ) )
   ( Signed: ... ) )
```

so that members of my group `banana-lovers` can obtain the RC5 decryption key, and thus read the posted multi-cast announcement.

Other more complicated protocols could be devised for this. For example, I could published the encrypted key with the document itself, so that a group member would have to ask the server to decrypt the key for him (rather than doing a name lookup).

8.9 Key Compromise

Although SDSI does not support CRL's, there is nothing to prevent someone from offering a service supporting "hot-lists" of principals whose keys have been reported as "compromised." When a user believes that his private key has been compromised, he can so inform the service. (His own servers can also offer this service.) His message to the service can be authenticated by a statement signed by the compromised key. The service maintains a group (called, say, `compromised-principals`) giving all such principals.

9 Conclusions

We have presented a simple yet powerful framework for managing security in a distributed environment, and hope that the perspectives and style shown here may assist others in building more secure systems.

Acknowledgments

We thank Carl Ellison and Wei Dai for many thoughtful comments on earlier versions of this paper.

References

[1] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, page (to appear), May 1996.

[2] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *IEEE Security Privacy*, 1987.

[3] Stephen T. Kent. Internet privacy enhanced mail. *Communications of the ACM*, 36(8):48–60, August 1993.

[4] National Bureau of Standards. Secure hash standard. Technical Report FIPS Publication 180, National Bureau of Standards, 1993.

[5] Ronald L. Rivest. The MD5 message-digest algorithm. Internet Request for Comments, April 1992. RFC 1321.

[6] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.