A Java Implementation of
Simple Distributed Security Infrastructure

by

Alexander Morcos

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 22, 1998

©1998 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author ____________________________
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by ________________________
Ronald L. Rivest
Thesis Supervisor

Accepted by _______________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

JUL 14 1998
LIBRARIES
A Java Implementation of
Simple Distributed Security Infrastructure
by
Alexander Morcos

Submitted to the Department of Electrical Engineering and Computer Science

May 22, 1998

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

ABSTRACT

Two Java packages have been written which contain classes to implement S-expressions
and Version 2.0 of the Simple Distributed Security Infrastructure (SDSI) specification.
Another package has been written, which contains classes that provide a graphical user
interface to the classes and methods in the other packages.

Thesis Supervisor: Ronald L. Rivest
Title: Edwin Sibley Webster Professor of Computer Science and Engineering
Acknowledgements

This work is dedicated to my family. Thanks to my advisor Professor Ronald Rivest and to my friends for their encouragement. This research was supported by DARPA Grant DABT63-96-C-0018.
Chapter 1

Introduction

The Simple Distributed Security Infrastructure (SDSI)[1] is a public-key infrastructure developed by Ron Rivest, Carl Ellison, and Butler Lampson. Public-key infrastructures provide frameworks for authentication of digital information. SDSI is a simple such infrastructure designed to facilitate passing of authority. This document describes an implementation of SDSI in Java. Three packages were built for this implementation. These packages together constitute a library of code, which implements the SDSI 2.0 specification.

SDSI is built on top of the language of S-expressions[2]. S-expressions are a structure used to represent data. This implementation also contains classes that provide for the parsing and creation of S-expressions.

Users of a public-key infrastructure issue digital certificates and sign them. These certificates usually contain a statement about another user in the system. In SDSI, certificates allow one user to give authority to another user, for example permission to read a file.

The Java implementation of SDSI provides classes that implement the fundamental SDSI objects, such as certificates and public-keys. These classes can be used to build application tools that work with the SDSI infrastructure. A certificate management tool
was built that allows for the creation of new certificates and the verification of certificate chains.

1.1 Design of SDSI

Public-key infrastructures provide a framework within which the tools of cryptography, authentication and encryption can be employed. SDSI is designed for the purpose of authentication. Using SDSI facilitates the process of passing authority, and proving and verifying authorization.

There are three major components of a public-key infrastructure.

- **principals** – these are the users of the system; the entities who make statements and about whom statements are made.

- **certificates** – these are the digital documents that contain a statement. In SDSI, certificates are used to transfer authority from one principal to another.

- **public-keys** – these fundamental tools guarantee authenticity. Principals are bound to a public/private key pair. A principal’s private key signs a certificate issued by that principal. The certificate is authenticated by any other user by verifying the signature with the issuing principal’s public-key.

A useful model to consider is a user trying to gain access to a resource. The user presents a chain of certificates to a server, which controls that resource. The server then verifies that the certificate chain indicates that some principal on the server’s access control list has passed authority to the user making the request, possibly through several intermediate principals. The server can then release the resource to the requestor.
Chapter 2 gives more detail on public-key infrastructures and how they are used. The chapter then concentrates on the SDSI system in particular, covering the design principles behind SDSI and explaining briefly how SDSI works.

1.2 S-expressions

The SDSI system uses S-expressions to encode its objects. S-expressions are data structures that facilitate the encoding of complicated data and provide a unique canonical form so the data can be signed and verified. The \texttt{sdsi.sexp} package is devoted to the implementation of S-expressions.

The classes in the \texttt{sexp} package provide the tools necessary to parse S-expressions from an input stream and encode them in a \texttt{Sexp} object. \texttt{Sexp} objects represent S-expressions. The \texttt{Sexp} object and its associated subclasses are explained in Chapter 3.

1.3 SDSI Objects

The \texttt{sdsi} package contains the classes that implement objects in the SDSI world, such as certificates, public-keys, signatures, and sequences. These classes have the functionality to perform operations that are needed in SDSI. The package also contains helper classes that are used within these main SDSI objects as well as classes used in the process of verifying a certificate chain. The \texttt{sdsi} package is described in Chapter 4.

1.4 SDSI Certificate Utility

The \texttt{sdsi.control} package contains the classes that implement the graphical user interface. This interface, the \texttt{SDSI Certificate Utility}, allows the user to create and use
SDSI objects. The *Certificate Utility* provides functionality for verifying certificate chains as well as issuing new certificates. It also maintains a database of stored certificates and other SDSI objects for a user. This package is discussed in Chapter 5.
Chapter 2

Design of SDSI

Version 2.0 of the SDSI specification is a merger of two separate public-key infrastructures. The Simple Public Key Infrastructure (SPKI), designed by Carl Ellison, and SDSI 1.0, designed by Ron Rivest and Butler Lampson, were merged into what is now SPKI/SDSI 2.0, which is referred to in this document as SDSI. SDSI currently exists as an internet draft of the Internet Engineering Task Force (IETF).

Section 2.1 discusses public-key infrastructures and SDSI as a public-key infrastructure. Two basic principles in the design of SDSI were that all name spaces would be local and that principals would be public-keys. Typically, principals are bound to public-keys through identity certificates. In SDSI, the principals actually are the public-keys. These design decisions are covered in Section 2.2. Section 2.3 explains how to interpret the meaning of certificates in SDSI.

2.1 Public-key infrastructures

Public-key infrastructures specify the way in which principals should be bound to public-keys. Commonly, principals correspond to real-world people, however this is not the case in SDSI. In SDSI, the correspondence between real-world people or organizations and the principals in the system is left to the human user.
Public-key infrastructures can be used either to facilitate secrecy by encryption, or authenticity by digital signatures. Depending on the algorithms which the public-keys are part of, they can be used for one or both of these purposes. Public-keys are used by someone in the system to verify a digital signature supposedly issued by the principal, or to encrypt a message to be sent to the principal. Private keys are used to sign a digital document that the principal issues or decrypt a message sent to the principal.

Public-keys in the SDSI world are represented either by a SDSI object that is that public key, or by a SDSI object that is a secure hash of that public key. Secure hashes are more efficient to transport than public-keys. An example of a SDSI public-key and a hash of that same key follow:

(public-key
 (rsa-pkcs1-md5
  (e #010001#)
  (n |AKUqiR3LCD399tvoh0F4MBQTNKhPun9EZxVNxp4/SJb1MUQBNL
      kzBwDhsfm0xCy8c/DWUiKlOz8qtAfGUvGv3wQNa2WficXgc/0K
      lWrulPFy/ulg5iEGDHzotI+LuASv9Pq9aOG2a52B3z3DgsO0/ky
      PwwJuoItf1Yw4PB|)))

(hash shal |AMY5MM9ge//AKBaARJEsx32w2c=|)

One of the fundamental objects of a public-key infrastructure is the certificate. A certificate is a digital document that makes some statement. Certificates are issued by someone who controls a public-key (meaning they have access to the private counterpart), and it is accompanied by a digital signature, usually by the same key.

There are two common types of certificates, authorization certificates and identity certificates. Identity certificates serve to make the binding between a key and a real-
world entity. Many of the issues in a public-key infrastructure revolve around trust. Essentially, a user of the system must trust some knowledge about which public-keys belong to which real-world entities.

Identity certificates are used in situations like the following scenario. Suppose Alice and Bob were users in a public-key infrastructure. Assume there is a key, $K_b$, that Alice knows belongs to Bob, and suppose Bob issues and signs a certificate stating that some other key, $K_c$, belongs to a third person, Carl. This certificate is an identity certificate. If Alice trusts Bob's judgement, she can now add to her trusted knowledge the fact that $K_c$ belongs to Carl. This is secure because Alice can use $K_b$ to verify Bob's signature on the certificate.

Many public-key infrastructure, such as X.509[3], deal with this concept of identity certificates and trust. They include such features as specifying rules for creating identity certificates and rules that establish how trust transfers. There are, however, many problems involved with such a system including how to bootstrap the system from its initial point where no one has any trusted knowledge.

Although identity certificates are important and can be issued in the SDSI system, SDSI does not concentrate on these issues. The problem of binding a public-key to a real person is not addressed. This process is assumed to require a human element and therefore can not be specified by a set of logical rules. Therefore an identity certificate in the SDSI system looks like the following and must receive human attention to have any meaning.
This certificate states that Carl Ellison is the keyholder for the key whose hash is given in the keyholder field. It is issued by the key whose hash appears as the issuer, and would commonly be accompanied by a digital signature that could be verified using the issuing key. The meaning and importance of this certificate is not specified at all. The only thing the SDSI system ensures is that its authenticity can be checked, if it has an accompanying signature. A user must make his own judgements as to who really controls the key.

Authorization certificates pass authority between principals. A certificate can state that the issuer gives some other principal the right to access a resource, such as reading a file. An authorization certificate in the SDSI system looks like the following:

(cert
 (issuer (hash md5 |Z4a6hysK/0qN0L5SfkcJFQ==|))
 (subject
  (public-key
   (rsa-pkcsl-md5
    (e #010001#)
    (n
     |AJmYpAYaZVPEYfI313kIatOqMkqtrrbfGebofx47gyracSNgWIkU52atwAf
      PQwCJ3ZiI3iZUt9CZbqQmqCdeuyhRvCkDgkJMcJeY5e/WM96EfamaT5
      LHmixNcoJESa7oR2Z+TrVfmcNjpuY5SfZjG6VycFmc9tnqBij|)))
   (propagate)
   (tag (read foo))))
)
This certificate states that principal whose public-key is given by the hash in the issuer has given the authority to "read foo" to the principal whose public-key is given as the subject. Unlike the identity certificate above, this certificate has definite and explicit meaning in the SDSI system. It can be combined with other certificates to create a chain of authorization, and it can be verified when accompanied by a valid signature. This will be explained in more detail later.

2.2 SDSI design issues

Two of the fundamental design points of SDSI were to make the principals of the system actually be the public keys, and to have only local name spaces. By structuring the system this way, SDSI avoids many of the complications that plague systems like X.509.

In SDSI, the principals are public-keys. Therefore, identity certificates are never needed to check the binding of a principal to a public-key. In order to verify a chain of certificates, a system only needs to look at the certificates in the chain. These certificates pass authority either to other public-keys or to names, which can be securely resolved to public-keys. It is important to note that names in the SDSI system do not violate the rule that all principals are public-keys. This fact will be made clear later.

Names in SDSI are defined only locally. If some certificate is issued, which has a name as a subject, then that name has no global meaning but is defined only by the principal whose name space it is in. The name space is determined by explicitly stating within the name the principal whose name space the name is in. If the principal is left out the name is regarded as in the name space of the issuer of the certificate.
A name can only be bound to a key by a name definition certificate issued by the principal controlling the name space. This protects against the burden of having to start up a global name hierarchy before the infrastructure can become useable, and it also helps to protect against competing hierarchies by giving complete control of name definition to each principal.

It should be noted that the identity certificate given above does not use a SDSI name. The identity certificate is not part of the verification process in the SDSI system and can not be securely interpreted automatically.

2.3 5-tuples

A fundamental concept of the SDSI system is the 5-tuple. A 5-tuple is a logical construct, which consists of the five main values necessary for a link in an authorization chain. These are:

- **issuer**: a principal (public-key or hash of one) or a name.
- **subject**: a principal or a name.
- **propagate**: a bit specifying whether the authority may be further propagated.
- **tag**: the authority being transferred.
- **validity**: the period during which this 5-tuple is valid.

There are multiple types of certificates in the SDSI system, but when a certificate has either a name or a principal as a subject, it can be thought of as a 5-tuple. Certificates can not achieve the full generality of a 5-tuple, but when combined together they can achieve greater functional generality. Name certificates are 5-tuples whose issuer is always a
single name, whose propagate bit is always true, and whose tag is always (*) or
everything. Authorization certificates are 5-tuples whose issuer is always a principal.
The representation of these 5-tuples in the Java implementation is discussed in Section
4.6.5.
Chapter 3

S-expressions

SDSI objects are represented using the language of S-expressions. A good implementation of S-expressions is crucial for efficiency, because S-expressions are heavily used within the code. The `sexp` package contains the classes that implement S-expressions. This chapter is devoted to discussion of these classes. The first section provides a background explanation of S-expressions. Subsequent sections detail how S-expressions are represented, created and used.

The use of S-expressions led to a natural layer of abstraction for the implementation. The `sexp` package was written to abstract away the implementation of S-expressions from the implementation of SDSI objects. The package contains classes that represent S-expressions and can parse them from input streams. Functionality includes returning an S-expression in various forms and accessing the data contained within an S-expression.

3.1 Introduction to S-expressions

S-expressions allow data to be represented in a variety of formats that are useful for different purposes. These representations include short forms used for transport and forms that can be formatted for use in human interaction. S-expressions also have a unique canonical representation that is important for signing and hashing. SDSI objects
can be displayed as S-expressions in readable form and when operations need to be performed on them, they can be processed in canonical form. Below are two examples of a simple S-expression. The first example is displayed in readable form, and the second example is in canonical form. Both examples encode the same S-expression.

(S-expression
 (example "Alex Morcos"))

(12:S-expression(7:example11:Alex Morcos))

3.2 Representing S-expressions

Sexp objects represent S-expressions. The Sexp class provides the general framework for S-expressions and is the top of the class hierarchy. The Sexp class, an abstract class, is subclassed by SexpString and SexpList. These subclasses implement the two types of S-expressions, byte-strings and lists of simpler S-expressions. The SimpleString class provides much of the implementation and is heavily used by SexpString. Data is actually stored in SimpleString's.

S-expressions are always represented in Sexp objects in canonical form. None of the formatting for readability is stored. In fact, the parentheses that determine lists are not stored either, because the list structure is represented by the hierarchy of Sexp objects.
3.2.1 SexpList

The SexpList class represents S-expressions that are lists of simpler S-expressions. Its data representation is simply a Java Vector. This Vector contains only Sexp objects, but a Vector was chosen over an array because of the way SexpList's are constructed. When the SexpList constructor is called on an input stream, it doesn’t know how many S-expressions are contained in the list, so the newly created Sexp objects are just added one by one to the Vector.

3.2.2 SexpString

The SexpString class represents S-expressions that are just byte-strings. The SexpString class takes advantage of the SimpleString class to actually store the data of an S-expression and truly represent a simple byte-string. The SimpleString is used because an S-expression byte-string is not necessarily just a byte-string but can be accompanied by a display-hint. Both the display-hint and the content are represented in canonical form by SimpleString’s in the SexpString class. Often, the display member, which points to the SimpleString encoding the display-hint, is just a null pointer.

3.2.3 SimpleString

Although S-expressions are represented by SexpString’s and SexpList’s, SimpleString’s really encode the data of an S-expression. SexpList’s only store
pointers to other Sexp objects and SexpString’s just have two pointers to SexpString’s.

SexpString’s have two data fields that are byte arrays. data contains exactly the bytes of the canonical form of the byte-string, without the preceding length. length_bytes contains the bytes of the ASCII representation of the length of the data array. Although it is not strictly necessary to store the bytes encoding the length, one design goal was to make it very efficient to retrieve the byte array representing the S-expression in canonical form. Storing the bytes encoding the length makes it possible to retrieve the canonical form without having to convert every time from an integer length to the ASCII bytes representing that integer.

3.3 Creating S-expressions

Sexp objects can be created in two ways. The simpler way is to use the constructors associated with SexpList and SexpString to create a new S-expression from some byte-arrays or strings. A SexpString can be created just by calling the constructor with a string or byte-array for the data and an optional string for the display-hint. A SexpList can be created just by calling the constructor with an array of previously created Sexp objects.

Parsing an S-expression from an input stream provides the other way to create a Sexp object. This method is very useful for reading an S-expression from a file or receiving it from some other application.
3.3.1 Parsing an S-expression

To parse an S-expression from an input stream, a PushbackInputStream is constructed from the input stream we wish to parse. Then, the Sexp.parse method is called with this stream as its argument. The Sexp.parse method examines the first byte of the input stream to see whether the S-expression about to be parsed is a list or a byte string. The method then calls the appropriate subclass constructor on the input stream and returns the result.

The SexpList constructor recursively calls Sexp.parse on the stream it is given and places each newly created Sexp object in the Vector that stores the members of the list. The SexpString constructor determines whether there is a display hint and then just calls the SimpleString constructor on the input stream, to actually parse out the byte-string.

3.3.2 Parsing a SimpleString

The SimpleString constructor is a sophisticated parser. The constructor takes an input stream and first determines in which form the S-expression is encoding by looking for the special characters that imply quoted string, hexadecimal, or base-64 encoded forms. It also checks for a length preceding the actual bytes. If the S-expression is not in one of the special forms mentioned above, the length is used to distinguish canonical and token encodings. The constructor then reads in the byte-string and converts it to canonical form, storing the resulting bytes. The bytes that represent the length are
calculated and stored also. The constructor ensures that the byte-string is a legally
encoded S-expression and throws an exception otherwise.

3.4 Using S-expressions

The Sexp class provides several useful class methods that are used by the other classes
in the Sexp package. These methods provide functionality such as testing whether
characters are of a specific type, such as legal base-64 characters or white space, or
converting between different types of character representation. One of these methods is
the toString method, which converts a byte array to a string that has those bytes as its
ASCII representation.

The Sexp class also provides several abstract methods, which are implemented
by both of the subclasses. These important accessor methods allow use of the Sexp
objects. The most important of these methods are the following:

- `getCanonLen()` returns the number of bytes in the canonical representation of this
  S-expression.
- `getCanonRep()` returns a byte array which contains the canonical representation
  of this S-expression.
- `toReadableString(int offset, int width, int last)` returns a
  String which contains this S-expression in readable format. Byte-strings are
displayed as token strings or quoted strings when possible and byte-strings that
contain unprintable characters are displayed in hexadecimal or base-64 encodings.
The arguments are parameters for the formatting indicating the left margin, the overall width, and the right margin respectively.

As mentioned previously, the `Sexp` objects were designed to make `getCanonicalRep()` very efficient. Constructing the canonical representation simply requires that the various byte-arrays be concatenated with occasional punctuation characters inserted, such as parentheses, brackets, and colons.

`SexpList` also contains a few methods that allow access to the S-expressions contained in the list.
Chapter 4

SDSI Objects

The implementation of SDSI objects constitutes the heart of the library. The sdsi package bundles this implementation code together. The sdsi package contains the classes that represent SDSI objects and the functionality that is necessary to work within the SDSI system.

This chapter covers the sdsi package and its implementation of SDSI objects in detail. Section 4.1 gives an overview of the implementation of SDSI objects and discusses how they fit in with the rest of the library. Section 4.2 covers the representation of SDSI objects as data structures. A few examples are given of how particular classes are used to represent their corresponding SDSI objects. Section 4.3 explains how SDSI objects are created from S-expressions. Section 4.4 deals with the cryptographic functionality of the library. Section 4.5 discusses how the Java exception handling model is used to provide a clean implementation of exception handling within the library. Finally, Section 4.6 addresses the issue of verification. Verification refers to the process of checking a request for validity. This process involves examining a chain of certificates, which the requestor has presented. The certificate chain should show that authority for the request has logically passed to the requestor. Section 4.6 looks at the role many different classes play in the verification process.
4.1 Implementation of SDSI Objects

Each type of SDSI object is represented by a class in the sdsi package. These objects range from such general top-level objects as certificates to specific parts of SDSI objects such as tags. The SDSIObjec object class is the base class for this hierarchy of SDSI objects. A number of methods are contained in this class that allow for use of the object and are inherited by all of the subclasses.

An application that uses this Java implementation, would be built based on the classes in the sdsi package. The application would create, manipulate, and use SDSIObjec object’s to perform whatever SDSI operations are desired. The user interface designed in the control package follows this approach and works with a database of SDSIObjec object’s that it creates and maintains.

The separation of the implementation of S-expressions and the implementation of SDSI objects allows SDSI objects to be viewed abstractly without concern for how they are represented as data structures. An application using SDSI objects could work only with the SDSIObjec object classes and be unaware of their representation as S-expressions. This separation allows future changes in the way SDSI objects are represented to take place without changing the code in an application that uses SDSI objects. For example, SDSI objects could just as easily be represented with XML as with S-expressions.

The separation is not entirely complete as can be seen by the use of the sexp package in the user interface. The separation can not be complete because it is necessary to use Sexp’s in the constructors of SDSIObjec object’s. In addition, the idea of signing a
SDSI object requires that there be a canonical way of representing it, which can be used for signature and verification purposes.

4.2 Representing SDSI Objects

Objects in the SDSI world are implemented by the `SDSIObj ect` class. In its simplest form, as a base class, this implementation consists of storing the S-expression that represents the SDSI object. The S-expression is stored as a `SexpList` in the variable `srep`. Subclasses of the `SDSIObj ect` class also contain other instance variables, which are often other SDSI objects. These other variables allow abstraction of the various parts of the object. Examples of these other variables can be seen in the sections below on the representation of certificates and keys.

When `SDSIObj ect`'s are created from an S-expression, the S-expression is parsed to determine the type of SDSI object it contains and the appropriate subclass of `SDSIObj ect` is then created. The S-expression is completely parsed to check for its validity as a SDSI object, and the various parts of the SDSI object, such as a tag or a subject, are also read in and parsed into their respective `SDSIObj ect`'s. These objects are often stored as variables within the containing `SDSIObj ect` class. This method is chosen over the possibility of just storing the S-expression and attempting to parse its various parts, as they are needed.
4.2.1 Memory Usage

Concerns about memory usage played a large role in designing the way SDSI objects would be represented. The ability to have instance variables, which refer to various parts of the SDSI object, was considered important. Otherwise, the full S-expression would have to be parsed every time an operation was performed with the SDSI object. Due to the possibly hierarchical structure of SDSI objects, this makes the repeated storing of the same information a concern.

An example of how this hierarchical structure can lead to very inefficient memory use is the case of a sequence. A sequence is a list of certificates, the sequence is a SDSI object and thus when implemented by the Sequence class (a subclass of SDSIObject)}
SDSIObject) it stores a SexpList which represents the S-expression of the sequence. The Sequence class also stores an array of the SDSIObject’s contained within the sequence, so that the individual parts of the sequence don’t have to be parsed into SDSIObject’s again and again. Each of these SDSIObject’s will also have an srep variable, which encodes the S-expression for that particular object. In turn, these objects might contain variables that are also subclasses of SDSIObject and store their own S-expression representations.

If we are not careful, however, we will be duplicating the storage of the same data. The entire S-expression has already been placed in memory once with the srep variable of the Sequence class at the top of the hierarchy. A very convenient way to refer to the same information again without duplicating it is with the use of pointers. Java does not allow the use of pointers. Fortunately, the use of classes implicitly involves the use of pointers, since objects in Java are passed by reference.

The srep variable in the sequence will have, as its representation, a Vector of other Sexp objects. When the member SDSIObject’s of the sequence are created, they will be created with these same Sexp objects, and so their srep’s will just refer to the same data structures that the SexpList of the Sequence refers to. No data will ever be duplicated within the storage of any one SDSIObject, however the same data may be referred to many times by different variables in the SDSIObject.
4.2.2 Representing Certificates

Cert objects represent certificates. The Cert class is an abstract class, which contains the methods and data that are common to both name certificates and authorization certificates. In addition to the S-expression, a Cert stores the values of the 5-tuple that the certificate encodes. The variables that store these values are a Tag, a Validity, a Subject (a marker interface which is implemented by all SDSIObject’s which can be subjects), a SDSIPrincipal for the issuer, and a boolean for the propagate bit.

There are two subclasses of Cert. Auth objects represent authorization certificates. Def’s represent name certificates. In a Def, a Name object is also stored; issuerName stores the name being defined by the certificate. The issuer of the name is always the same as the issuer that is represented by the SDSIPrincipal in the issuer field of Cert.

Both Auth’s and Def’s, therefore, provide quick access to the components of the 5-tuple they represent.

4.2.3 Representing Keys

Keys in this implementation are represented by several different classes:

SDSISecretKey, SDSIPrivateKey, and SDSIPublicKey. The implementation of public-keys is the most interesting, as public-keys are the backbone of the SDSI system and they are used far more often than private or secret keys.

Essentially, SDSIPrivateKey just holds the private key corresponding to a public key so that the entity controlling the principal can sign certificates and other
objects. Private-keys do not necessarily need to be viewed as SDSI objects since they will only be used internally and never passed around.

The **SDSIPublicKey** class implements public-keys. This class is subclassed into a different class for each type of public-key algorithm that might be used. In the current implementation, only **SDSIRSAPublicKey** exists as a subclass. A **SDSIPublicKey** does not actually store any additional information. When a **SDSIPublicKey** is created, it does some basic syntax checking on the S-expression and provides a set of methods that is useful for dealing with public-keys. The data that needs to be accessed is dependent on the public-key algorithm and is therefore stored in the appropriate subclass.

A **SDSIRSAPublicKey** stores as byte arrays the modulus and exponent of the public-key. Although it is not immediately apparent, even this does not cause duplicate bytes to be stored in memory. The implementation of a **SimpleString** is a byte array that contains the bytes of the byte-string being represented. These same byte arrays are pointed to again by the **modulus** and **exponent** variables in a **SDSIRSAPublicKey**.

It should be noted that **SDSIRSAPublicKey**'s provide an example in which further processing must be done when using the **SDSIObj ect**. The cryptographic functions that use RSA public-keys need to access the modulus and exponent as **BigInteger**’s. There are methods in the class which return these **BigInteger**’s but they are not stored as such, and must be recreated from the byte array’s every time they are needed. This decision was a tradeoff of processing time for storage space.
4.3 Creating SDSI Objects

SDSIObject’s are created using the SDSIObject.principalParse() method. This method takes a SexpList as an argument and then parses a SDSI object out of the S-expression represented by the SexpList. The parsing proceeds by analyzing the first element of the SexpList, which should be a SexpString, and using it to determine the type of SDSI object that is being encoded. The object type can be discovered in this way because all SDSI objects are represented as lists whose first element is a simple byte-string indicating the type of object. The constructor of the subclass that implements that SDSI object is then called on the SexpList and the newly created SDSIObject is returned.

The principalParse method also takes a SDSIPrincipal as an argument. If the object being created is an acl, a name, or a key-holder object, then this principal is included as an argument to the subclass constructor. If an appropriate principal is not known, null can be passed in as the argument. If the object being represented is not one of these three types, then the subclass constructor is called with only the SexpList as an argument, and the principal is ignored.

For acl’s this principal serves the purpose of identifying, in the Acl object, the principal that created the acl. The S-expression for an acl does not specify the creating principal, but is often useful information when working with an acl. For names, the principal is necessary because names may not be fully qualified. For instance, when a name appears in the context of a certificate, the issuer of the name may just be the issuer of the certificate. In this case, the issuer may not be specifically represented in the S-
expression for the name. However, the issuer information needs to be kept in the Name object, so it must explicitly be passed in as an argument to the constructor. The same logic applies to the key-holder object, because the subject of the key-holder might be a name.

In general, the constructors for SDSIObject's work by extracting the various parts of the S-expression and parsing them into their respective SDSIObject's, if necessary. The constructor verifies that the SexpList is a correctly encoded S-expression that represents the correct type of SDSI object. An exception is thrown if the SexpList can not be parsed correctly.

The degree of error checking varies depending on the SDSI object. For instance, in a SDSIRSAPublicKey, the byte-strings for the exponent and modulus are not actually checked to see if they correctly encode large integers that could be an exponent and a modulus.

4.4 Cryptographic Functions

As a security architecture, SDSI needs to use a number of cryptographic tools. The tools used in this implementation of SDSI are digital signature algorithms and secure hash functions. The Java Development Kit specifies a Java Cryptography Architecture (JCA), which facilitates the use of these types of functions. As of JDK 1.1.4, which was used for this implementation, there were still many features missing from the java.security package. However, specifications were available, in an early-access release, for the
improvements to the Java Cryptography Architecture that would be included in the release of JDK 1.2.

Cryptix[4] provides a suite of class which implement the International Java Cryptographic Extension in early release form. These classes are heavily depended on by this implementation for signature creation and verification. In addition to an implementation of the RSA algorithm, the Cryptix code specifies the \texttt{RSAPublicKey} and \texttt{RSAPrivateKey} interfaces, which will be included in the new JCA. The \texttt{SDSIRSAPublicKey} and \texttt{SDSIRSAPrivateKey} classes implement \texttt{RSAPublicKey} and \texttt{RSAPrivateKey} respectively. These interfaces provide function prototypes for methods to access the modulus and exponent of an RSA key. The SDSI RSA key classes provide implementations of these methods.

Cryptix works within the security architecture of Java. This security architecture allows for the installation of providers that implement various algorithms. In theory, the code in this library does not require that Cryptix be installed on the system. However, a security provider that provides implementations of the algorithms that are used is required. Details on how to set this implementation up, including cryptographic functionality, are provided in Appendix A.

The current implementation of SDSI uses the MD5 and SHA-1 hashes and the RSA/PKCS1/MD5 and RSA/PKCS1/SHA-1 signature algorithms. The MD5 and SHA-1 hashes are provided in the default Java Cryptographic Extension, and, as mentioned before, Cryptix provides an implementation of the RSA algorithm. For additional algorithms to be added to the SDSI world, very little work is required in updating the
**sdsi** package, as long as the algorithm is supported within the Java Cryptography Architecture.

First, if the currently installed providers do not implement the desired algorithm, a provider that does will have to be installed. When a new hash algorithm is added a few lines like the ones below will have to be added to translate between the SDSI name for the algorithm and the Java name. These lines appear in the code for creating hash objects.

```java
if (algorithm.equals("md5")) {
    md = MessageDigest.getInstance("MD5");
}
```

If the new algorithm is a signature algorithm then a new **SDSIPublicKey** subclass will have to be coded which can correctly parse the syntax of the new public-key and also implements the correct public-key interface within the Java Cryptography Architecture. Then, as with the hash algorithm, a couple of lines similar to below will have to be added to recognize the new type of public key in the **SDSIPublicKey.create** method.

```java
if (alg.equals("rsa-pkcs1-md5"))
    return new SDSIRSApublicKey(l);
```

No other changes will have to be made because the default Java **Signature** and **MessageDigest** classes should be able to provide all the necessary functionality for any algorithm, once the provider is installed.

This implementation was not designed with the security of the actual application in mind. Currently, private keys are stored in the clear in files, which then are read in to establish the principal who will be controlling the SDSI system and issuing signatures.
4.5 Exception Handling

Java provides a very clean and easy to use model for exception handling. The implementation of SDSI takes advantage of this model in its own exception handling.

The implementation defines three main types of exceptions:

- **SexpException** – The methods that create Sexp’s throw this exception. When it occurs, this exception indicates that the input stream being processed does not contain a valid S-expression. Conditions that can cause a SexpException to be thrown range from illegal characters to mismatched parentheses.

- **SexpParseException** – When a Sexp is being parsed into a SDSIObject, this exception can be thrown. Its occurrence indicates that the Sexp being parsed does not correctly encode a SDSIObject. This incorrect encoding can happen if required fields in the representation are missing or have illegal values. Many of the methods of SDSIObject’s, most notably the constructors, throw this common exception.

- **SDSIException** – This exception occurs infrequently. Cases that are not covered by either of the other two exceptions are covered by SDSIException’s. An example of a situation in which this exception can be thrown is when a valid S-expression correctly encodes a SDSI object, but that object, or its content may not make sense in the current context. For instance, a Name requires that it have an associated creating principal, although the S-expression representation of a name is not required to contain a principal. When a new Name object is created, the principal
must be either explicitly given or contained in the Sexp. If it is not, a

SDSIEException will be thrown.

In addition, a VerificationException is defined for use when the attempted
verification of a sequence fails.

All of the exceptions are subclassed from the Exception class. They all have
constructors that can take a String as an argument. An application that uses the sdsi
package or the sdsi.sexp package can catch these exceptions and signal to the user
that an error that has occurred. In most cases, these exceptions do not signify a fatal
error, and there is no need to terminate the controlling application.

Many other exceptions are also thrown and caught in the code for this
implementation; they are, for the most part, transparent to the user. There are several
exceptions that, if they occur, signal a fatal error that would prevent the packages from
working properly. Most notably, the UnsupportedEncodingException can be
thrown when trying to translate between characters and the bytes that encode them using
the 8859_1 encoding method. This encoding is required to process bytes as ASCII
characters, as needed for S-expressions. All systems should support this encoding and in
most cases this exception is caught and ignored because it should not occur.

4.6 Verification

The SDSI system has two primary functions: proving and verifying. The concept of
proving consists of a principal, the prover, generating, from some database of certificates,
a chain of certificates and accompanying signatures which show a logical passing of
authority from some initial principal to the prover. This process is usually done with an acl entry in mind and a specific request that matches the tag on the acl entry. The chain proves that authority passes from the subject of the acl entry to the prover.

This aspect of SDSI was not implemented by this Java implementation. Jean-Emile Elien has implemented a certificate discovery system, which from a database of certificates constructs such a chain if possible[5]. The code in this library should eventually interface with his work so that user of the SDSI Certificate Utility, discussed in Chapter 5, can create such chains.

Verification complements the task of proving. Verification consists of receiving a chain of certificates and signatures and verifying that it does indeed pass the authority requested from the subject of an acl entry to the prover. This functionality is provided with this implementation. The basic step in the verification procedure reduces two 5-tuples into a single 5-tuple. This process yields a 5-tuple whose interpretation is the same as the combination of the previous two 5-tuples.

This concept of a 5-tuple is embodied by the Tuple class. Tuple's can be created from Cert's and then two Tuple's can be combined to give the Tuple that represents their reduction. A Sequence can be processed to yield the Tuple that results from the reduction of all the certificates in the sequence.

The use of an object oriented system made the task of writing the verification code much simpler. Many rules govern whether or not two 5-tuples can be reduced, and what the values in the resultant 5-tuple are. These rules are however largely independent
for each of the variables in the 5-tuple. This independence makes it very easy to code the 5-tuple reduction in several layers of abstraction.

The Tuple class contains the general reduction code, but many of the combination rules are coded as methods of the objects in the Tuple. The parts of a Tuple are:

- the propagate bit: a boolean.
- the validity field: a Validity.
- the tag: a Tag.
- the issuer: a SDSIPublicKey (and optional Name).
- the subject: a SDSIPublicKey (and optional Name).

The combination rules for the propagate bit are very simple, if either boolean is false, then the resultant boolean is false.

The other objects Validity, Tag, SDSIPublicKey, and Name are discussed below to show the methods they provide for use in reduction. Then, the Tuple class and the way these methods are put together is explained. The final section describes the Verification class, which implements the process of taking a Sequence and finding the Tuple that results from processing all the certificates in the sequence. A Verification can also check this resultant Tuple against an acl and a request.

### 4.6.1 Validity

The Validity object represents the fields in a certificate or an acl, which specify the time period for which the object is valid. Currently, the implementation does not support the online-testing aspect of the valid field but only the not-before and not-after dates.
This object is not implemented as part of the **SDSIObj**ect hierarchy. That decision was made because the valid information is often not represented by a single S-expression but by multiple S-expressions. Implementing **Validity**’s as **SDSIObj**ect’s would violate the clean representation of a **SDSIObj**ect by a single **SexpList**.

The Java Date object was the clear choice for the representation of the not-before and not-after fields. A **SimpleDate**Format was created so that dates in the SDSI syntax could be created and parsed. The methods required for reduction merely update the valid field to reflect a more recent not-before date or a less distant not-after date.

### 4.6.2 Tag

Tags are one of the more complicated concepts in the SDSI world. Tags are implemented by letting the **Tag** class be a **SDSIObj**ect and then creating a helper class called **TagExpression** which implements the functionality of the tag. The **Tag** class is merely a container for the **TagExpression** and the **SexpList** that represent the tag.

**Tag**’s were implemented in this way because the hierarchy of classes needed to implement the functional possibilities of a tag is rather extensive, and it didn’t make sense for all of those subclasses to be part of the **SDSIObj**ect hierarchy.

There are two types of tags in SDSI. There are simple or star-free tags such as

(read foo)

and star tags such as

(read (* set foo bar))

or

(allow (* range numeric 1 10)).
Star tags authorize a range of permissions and star-free tags authorize only a single thing. The intersection of a star-free tag and any other tag can only be either that same star-free tag or null.

For the purposes of reduction, Tag's have an intersect method. A tag can be intersected with a star-free tag and a boolean return value will indicate whether the star-free tag is allowed by the other tag. The implementation of this within TagExpression's is a little bit more general. The intersect method instead returns the resulting tag, when two tags are intersected. Currently, the implementation requires that the argument tag be star-free, so the result is either the same tag or null. However, much of the code can be changed only slightly to reflect a more general intersect functionality that can compare two non-star-free tags and give the true intersection of them.

There is no clear need for this more general intersection in the way SDSI is currently used. However, if the concept of a result certificate, the effect of encoding a 5-tuple that is the reduction of a certificate chain back into a certificate, becomes more prominent, then it will be important to have non-simple intersection results.

4.6.3 SDSIPublicKey

This class provides a method that can compare another public key to the current one. The argument can be a principal in either form, a hash or a public-key. If it is a public-key, then true is returned if the two keys are equivalent. If it is a hash, then this key is hashed by the same hash function and true is returned if the two hashes are equal.
4.6.4 Name

The Name class provides several methods that are useful in certificate chain reductions. Often, the name, in the issuer or the subject of the 5-tuple, expands, contracts or is otherwise changed when the 5-tuple is combined with another one. The following methods are useful:

- `sameNamesAs` – tests to see if two names are the same, assuming they both are in the same name space.
- `prefixNamesOf` – tests to see if the names in this name are a prefix of the names in another name, again assuming they are both in the same name space.
- `expandNames` – expands the names in this name by some other names. The use of this method will become clear in the next section.
- `replaceNames` – replaces some prefix of the names in this name by a list of other names. This method is used when part of the subject is rewritten by a name definition certificate.

4.6.5 Tuple

The Tuple class implements the concept of a 5-tuple. With the methods created in the classes above, the Tuple class becomes relatively simple. The implementation of Tuple essentially consists of a constructor and the method `add` which combines two Tuple's. The constructor takes the values of the 5-tuple from a certificate and fills in the missing fields of propagate and tag for a name certificate. The `add` method of a Tuple takes another Tuple as an argument. The method works by modifying the
Tuple to be equivalent to the combination of its 5-tuple with the argument 5-tuple. The combination is generated from the rules given below.

The first three steps of the combination process are to logically and the propagate bits, update the validity field, and intersect the tags. These steps are straightforward given the methods discussed above. Then, the new issuer and subject must be created.

The rest of this section describes the rules for combining issuers and subjects. When two 5-tuples are combined, the 5-tuple that is given by the argument Tuple is considered the second 5-tuple in the reduction, and the 5-tuple encoded by the Tuple whose add method is used is considered the first 5-tuple.

If the argument 5-tuple has a tag, that is, it was created with an authorization certificate, then the only option is for the subject of the first 5-tuple and the issuer of the second 5-tuple to be exactly the same. The new 5-tuple has the original issuer and the subject of the second 5-tuple.

If the argument 5-tuple has only name-resolution properties then there are three possibilities:

- The subject name of the first 5-tuple can be the same as the issuer name of the second 5-tuple, in which case the subject of the resultant 5-tuple is the subject of the second 5-tuple. The issuer remains the same.

- The issuer name of the second 5-tuple can be a prefix of the subject name of the first 5-tuple, in which case the subject of the resultant 5-tuple is the subject of the first 5-tuple.
tuple with the matched prefix of the subject replaced by the subject of the second 5-tuple. The issuer remains the same.

- The subject name of the first 5-tuple can be a prefix of the issuer name of the second 5-tuple, in which case, the issuer name is expanded by the part of the issuer name of the second 5-tuple that is longer than the matched prefix. The subject of the resultant 5-tuple is the subject of the second 5-tuple. This expansion of issuer name happens when the name, which the certificate chain is eventually trying to reduce, is longer than the name in the issuer so far. This expansion can only happen if the certificate chain has not seen a Tuple that performs an authorization yet. There is a bit which keeps track of that fact.

4.6.6 Verification

The **Verification** class provides the final step in the verification process. This class implements the grammar of sequences. With the addition, to the grammar, of the possibility of defining sub-sequences and reusing them within the verification process, it becomes very useful to reduce a sequence to a 5-tuple. This process is implemented in the **Verification** class.

The sequence reduction process could be abstracted into the **Sequence** class or, in keeping with the way single certificates are handled, implemented in the **Tuple** class. However, processing a sequence requires considerably more machinery that processing a

---

1 Actually, name certificates are represented by **Tuple**s with (*) as a tag, so there is actually an extra bit to determine whether a **Tuple** is equivalent to only name certificates.
A sequence can refer to hashed certificates in addition to hashed keys. A sequence might contain subsequences to process and name, or previously processed subsequences might be referred to.

The **Verification** class provides a framework for this machinery and stores the necessary state information. The verification process must have access to the hashtables that match key and certificate hashes to their respective objects. The necessary hashtables can be provided by building the tables from scratch as the sequence is processed and hash opcodes are encountered. However, external hashtables can also be provided to start with, that are then augmented by any hashes in the sequence. The **Verification** class also contains a lookup table for matching defined names to their respective **Tuple**s.

This framework also allows for the creation of 5-tuples from certificates. The **Tuple** class constructor requires that the public-keys of the issuer and subject or the keys that start the issuer and subject names are given with the certificate. These keys must be given because the **Tuple** being created will have no access to external state to determine what key a hash is of. These values can be looked up in the tables in the **Verification** class and then the **Tuple** can be constructed.

Since there is no machinery for intersecting two tags and determining the actual intersection, it is necessary to have in mind the eventual request when processing a sequence. Thus, the 5-tuple that corresponds to the first certificate in any sequence is
always generated with the final request as an argument\(^2\). This tag then becomes the tag of the 5-tuple, assuming that it intersects with the tag on the certificate. Then as the chain is reduced and each 5-tuple is combined, the resulting 5-tuple will always either have the original request or null as its tag.

The \texttt{Verification} class also contains a method that facilitates the actual verification of the chain. That is, once the sequence has been reduced to its equivalent 5-tuple, the 5-tuple must then be checked against an acl and a signed request.

\(^2\) Currently all 5-tuples are created in this way. This is ok, because it is necessary and sufficient that each of the 5-tuples have tags that intersect with the final request.
Chapter 5

SDSI Certificate Utility

The \texttt{sdsi.control} package is a major component of this implementation. The package implements an application called the \textit{SDSI Certificate Utility}. This application provides an interface to the classes that implement SDSI objects and a framework within which to use these objects. The \textit{Certificate Utility} has a graphical interface designed with the Java AWT.

The \textit{Certificate Utility} was designed as a stand-alone application as opposed to an applet. Although the SDSI system might be very useful from inside a web-browser, to provide authorization for web-page access, the program is too complicated to be an applet. The application retains a lot of state information, uses memory extensively, and has a rather complex interface. In addition, designing the \textit{Certificate Utility} as an applet would have raised many security issues with file access.

Originally, a shell program was built to allow low-level access to the SDSI classes. The shell provided commands to read in SDSI objects and to manipulate them by performing actions such as signing and hashing. The shell was not kept up to date with the progress of the SDSI code, however, and does not currently function correctly. Due to the object-oriented structure of the code, the shell primarily translated between a command string and a call to one of the SDSI objects. Therefore, the shell was an unnecessary component of the implementation.
5.1 Interface

The interface to the SDSI Certificate Utility consists of a main window divided into three panels. The Certificate Utility also has a menu bar with several menus and a status field that gives feedback. The left panel, which takes up half of the display, is the display window. This window displays SDSI objects in a readable form. Care is taken to keep the bounds of each display line under the width of the window.

The top right panel, the object list, is also primarily used for display. The object list displays a list of SDSI objects. The objects are represented in the list by a unique number as well as a short string. The view menu on the menu bar can be used to choose which types of SDSI objects are displayed in the list. Clicking on an object in the list causes it to display in the display window. Double clicking on an object causes it to be selected for an operation being performed in the functional panel.

The bottom right panel contains the functional panel. Depending on the operation being performed, this panel displays a varying set of controls. These controls are used to customize the operation being performed. Most often, this panel provides fields to fill in information that is used to construct a new name or authorization certificate.

5.2 Design

The design of the graphical user interface proceeded quite differently from the design of the SDSI classes. The code was not divided into many classes which could be separately implemented and provide different levels of abstraction. The single class approach was chosen partly because all of the code was part of the same general environment and partly
for the sake of convenience. The Certificate Utility was designed using Symantec Visual Café. In Visual Café, the Java AWT controls were much simpler to use when all of the code was part of one class. This approach probably makes for more difficult to understand code and less reusability, but it was necessary due to time constraints and the often-changing design plans for the SDSI Certificate Utility.

Several helper classes were designed to go along with the SDSIMainFrame class. In general, these helper classes are used to pop up dialog windows or perform other maintenance. However, there is one other important class. The data structures in the Certificate Utility were implemented by the CertCache class. This class implements the cache that stores all certificates as well as public-keys and other SDSI objects. It is discussed in the following section.

The Certificate Utility is initialized by loading a private/public key pair that is controlled by the user. The user may then maintain his own database of certificates and other SDSI objects. These certificates and objects can be combined to create a certificate chain, be used to facilitate the creation of new objects, and be passed around to other members of the SDSI system. The user can save his database as a certificate cache and then on subsequent uses of the system can chose to load that cache.

The SDSIMainFrame class then, has only a few data structures, a certificate cache, a current object (this is the object in the display window), and a Verification (this is stored so that various parts of the Verification can be initialized at different times and then it can be checked).
5.2.1 The Certificate Cache

The certificate cache stores certificates that a given principal has accumulated. The cache is implemented by a series of Java hashtables. There are two sets of hashtables. In each set of hashtables, there are several hashtables that are used for different types of objects. The first set of hashtables contains the object tables. These tables have each SDSI object in the cache stored once\(^3\) as a value. The key corresponding to that value is a unique number, an index, assigned to that SDSI object. The second set of hashtables contains the hash indexing tables. These tables map each of the different SDSI hashes\(^4\) of an object to the index for that object. To retrieve an object, when its hash is known, the hash is used to look up the correct index in the hash indexing tables, and then the resulting index is used to look up the actual object in the object tables.

The bulk of the certificate cache consists of six object tables and six hash indexing tables. Each object table is paired with a corresponding hash indexing table. There are four of these table pairs for certificates, to handle name and authorization certificates separately, as well as to handle certificates created by the cache owner separately from those created by some other principal. There is one pair for keys and one for all other SDSI objects. In addition to these six pairs of tables, the certificate cache maintains the public and private keys of the owner, and a hashtable of names for keys, the

---

\(^3\) Note that each SDSI object is only stored once among all of the object hashtables, it is not stored once in each one.

\(^4\) There is a different hash and corresponding key/value pair for each hash algorithm the implementation is aware of. This number is currently two, MD5 and SHA-1.
name table. Finally, there is also a hashtable of objects that are displayed, the display table. The display table is useful for displaying objects with a short descriptive string, as in the object list. The name table and the display table will be explained shortly.

There are two methods to add objects to the hashtables. Whenever an object is added to the certificate cache, the object is assigned its index and added as the value with the index as the key to the appropriate object table. Then, all known hashes are applied to the object and the resulting hashes are added as keys with the index as the value to the hash indexing tables.

One method adds a certificate and its signature, they are stored as a pair in fact, to the appropriate pair of certificate tables. The hash is taken only of the certificate. Thus whenever, the certificate is referred to by hash value in a sequence it can easily be looked up. Certificates are only added to one of the four pairs of certificate tables when they are accompanied by a signature. A certificate without a signature has no validity and can not be used when creating a chain.

Whenever a certificate is signed in the utility, it is automatically added to the correct tables. In addition, whenever a hash opcode is encountered in a sequence with a certificate and a signature, the pair is added to the correct tables.

There is also a method that takes a generic SDSI object and adds it to either the public-key tables, if the object is a public-key, or to the miscellaneous objects tables. Similar to the case with certificates, whenever a public-key is encountered it is automatically added to the correct tables. Public-keys are automatically stored to improve the database of keys that are recognizable by hash.
One of the nicer features of the certificate cache is the name table; this hashtable maps a public-key to a string that serves as its name. Whenever a name certificate is added to the certificate tables, an entry in the name table is created, if possible. A name certificate which defines a name as another name may not be useful for creating a new entry in the name table, because it may still be unknown which key that name reduces to. A simple example where an entry can be added to the table follows. If K0 owns the cache and issues a name certificate K0 bob => K1, then the entry (K1, bob) is added to the name table.

The name table helps the user work with names whenever possible. The system uses the name table to determine what short string to display to represent a key. Currently the table only stores simple key-name bindings in the owner's own name space. However, extensions would allow a greater flexibility in naming, e.g. entries such as (K2, bob's alice). The system also removes bindings from this table whenever the corresponding name certificate is removed from the certificate hashtables.

The final part of the certificate cache is the display table that maps short descriptive strings to their corresponding SDSI objects. This table was specifically designed for use with the Certificate Utility. The certificate cache has methods to add a particular type of object to the display table such as all authorization certificates or all SDSI objects. These objects are retrieved from their respective hashtables, assigned a short descriptive string, and placed with this string as their key in the display table. The keys to this hashtable are used as the items that are displayed in the object list. When the
user selects an item from the object list, the actual SDSI object can be retrieved by finding the corresponding value in the display table.

The short descriptive string provides a less than one line description of the object using names from the name table whenever possible to represent a key. For instance, a key that happens to be bound to the name “Bob” in the owner’s name space would be listed as “[Key] Bob”, whereas the name certificate that defines Bob would be listed as “Bob”. Moreover, a certificate from Bob authorizing a key without a name binding to do something might look like “Bob #(read foo)# K-78213”. Clicking on any one of these items will, of course, display the actual SDSI object.

The certificate cache implements the Java Serializable interface. This allows the entire certificate cache to be saved to a file and then recreated at another time. Future improvements to the certificate cache would be to actually implement it as a file and scan it while still on disk. The size of the certificate cache in a real application may become too prohibitive for memory.

5.3 Functionality

The SDSI Certificate Utility provides an interface that allows a user to perform many useful tasks in the SDSI environment. There are four fundamental jobs that a user or a server must accomplish, for SDSI to work as a public-key infrastructure. The server must be able to verify a certificate chain against an acl and a request. The user must be able to maintain a database of certificates, issue new certificates, and construct a chain of certificates that authorize a specific request. The last of these tasks is the problem of
"Certificate Discovery" and is the subject of Jean-Emile Elien’s thesis. See [4]. The remaining three jobs are implemented by the Certificate Utility.

To perform one of these tasks the user selects a menu item from the menu bar. The different menus and the functions associated with their items are:

- **File:** This menu allows the user to read S-expressions, which represent SDSI objects, in from files, and to save S-expressions back to files. The user can also load a new identity and certificate cache by choosing files which contain the appropriate information or save the current certificate cache to a file.

- **View:** This menu allows the user to choose a specific class of SDSI objects to list in the object list.

- **Object:** This menu allows for object manipulation. The currently selected object may be hashed, signed, or removed from the certificate cache. In addition, the user can choose to create a new object by typing in an S-expression by hand.

- **Certificates:** This menu allows the user to create new name certificates or authorization certificates. The controls associated with these functions allow the user to choose a subject, select before and after validity dates, and choose a tag or select a name to define as appropriate.

- **Keys:** This menu provides a menu item that allows the user to create a new matching public-key/private-key pair.

- **Verify:** This menu allows the user to select a sequence to verify against an acl and a request.
With these menus and the controls associated with their items, the user can perform any of the previously mentioned tasks. Interfacing to the certificate discovery functionality should not be a difficult task. This will lead to full functionality of the Certificate Utility.
Appendix A

Using the Java Implementation

This Java implementation of SDSI was developed primarily on a Pentium 100 running Windows 95. The work was done using the Java Development Kit 1.1.4 (JDK) from Sun and with the help of Symantec Visual Café. The implementation uses no native methods and therefore, in principle, should run on any platform to which the Java Virtual Machine (JVM) has been ported. The implementation has been tested on Windows 95 and Solaris platforms. Some problems exist with particular X Windows systems but these problems are probably due to the implementation of the JVM and not solvable from the code in this implementation.

Another implementation of the SDSI 2.0 specification was done by Matthew Fredette. He designed a library in C to implement SDSI objects and functionality and has built a web-based user interface. Information on that implementation is available at [6]. The C implementation is interoperable with this implementation.

The classes and source for this SDSI implementation can be found at [7], and the documentation can be found at [8]. The code is available as compiled classes or as source files. The class files were compiled using the Sun Java compiler from the JDK. The source code can be compiled again if needed, and it should compile without any deprecation warnings under a Java 1.1.4 compiler. The code does contain features from the Java 1.1 specification including the new AWT model.
In order to use the classes of this SDSI implementation, the Java 1.1 classes should already be installed, as well as a Java interpreter. The path that the SDSI class files are placed in must be in the classpath or added to it. At this point the implementation is usable and the *SDSI Certificate Utility* can be run by executing “java sdsi.control.SDSIMainFrame” on the command line.

As mentioned earlier, for the implementation to be completely functional, security providers must be installed which provide implementations of the cryptographic and hash functions that are used. In the current implementation, the only algorithms that need to be supported are the MD5 and SHA-1 hashes and the RSA/PKCS1/MD5 and RSA/PKCS1/SHA-1 signature algorithms. The standard Sun security provider supports the MD5 and SHA-1 hashes. The RSA signature algorithms however must be implemented by another security provider. Cryptix provides implementations of these algorithms. The Cryptix classes can be found at [4] along with documentation and installation instructions.

In addition to supporting the RSA algorithms, the Cryptix code defines the `java.security.interfaces.RSAPublicKey` and `RSAPrivateKey` interfaces. These interfaces are part of the early access specification for Java 1.2 and are used for the RSA code in Cryptix. These interfaces must appear in the classpath in order for the implementation to work correctly. Currently this means that the RSA Cryptix classes\(^5\) must be in the class path. Later, these interfaces should be a standard part of the

\(^5\) Located in RSA_0-0-9.jar
classpath and it will only be necessary for Cryptix or another security provider to be installed, no changes to the classpath will be required.

Finally, the SDSI Certificate Utility attempts to load the Cryptix provider when it is first run. This dynamic loading of the security provider is done so that systems on which the user doesn’t have permission to install a new provider for the Java environment can still use the implementation. For this dynamic loading approach to work, the Cryptix provider class\(^6\) must be in the classpath. If a security provider has already been installed which implements the necessary algorithms, this dynamic loading is not necessary.

\(^6\) Located in SPT_0-0-9.jar
Bibliography


   http://theory.lcs.mit.edu/~rivest/sexp.html


   http://www.systemics.com/software/cryptix-java/


   http://theory.lcs.mit.edu/~cis/sdsi/sdsi2/java/SDSI_Java_Intro.html

[8] SDSI 2.0 Java Documentation.
   http://theory.lcs.mit.edu/~cis/sdsi/sdsi2/java/packages.html