An Implementation of SDSI - The Simple Distributed Security Infrastructure

by

Matthew H. Fredette

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

[illegible]

Author ....
Department of Electrical Engineering and Computer Science
May 20, 1997

Certified by ....
Ronald L. Rivest
Edwin Sibley Webster Professor of Computer Science and Engineering
Thesis Supervisor

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

An Implementation of SDSI - The Simple Distributed Security Infrastructure

by

Matthew H. Fredette

Submitted to the Department of Electrical Engineering and Computer Science on May 20, 1997, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

A library has been designed and written in C that implements Version 1.0 of the Simple Distributed Security Infrastructure (SDSI) specification. A SDSI server program and other relevant tools, including a command line "shell" for issuing SDSI commands, have also been written in C and make use of the library, which provides a rich set of functions for programs that want to use SDSI.

Thesis Supervisor: Ronald L. Rivest

Title: Edwin Sibley Webster Professor of Computer Science and Engineering

Acknowledgments

This is dedicated to my family and friends. This research was supported by DARPA Grant DABT63-96-C-0018.

Contents

1 Introduction **8**

1.1 The SDSI library 9

1.2 The SDSI server 9

1.3 Supporting tools 9

2 The SDSI Library **10**

2.1 General library framework 12

2.1.1 Fundamental data structures 12

2.1.2 The library context 14

2.1.3 Objects' current principals 16

2.1.4 Reserved words 17

2.1.5 Error codes 18

2.1.6 Principal definitions 18

2.1.7 Files 19

2.2 Memory management and error handling 20

2.2.1 Global state 22

2.2.2 Prologue and epilogue macros 23

2.2.3 Memory management 24

2.2.4 Error catching and throwing 27

2.3 Cache management 28

2.3.1 Localities 29

2.4 Object evaluation and manipulation 32

2.5 Protocol primitives 35

2.5.1 Transport mechanism 36
2.5.2 Protocol state 37
2.5.3 The protocol functions 38
2.6 Cryptography functions 39
2.6.1 Key profiles 40
2.6.2 Cryptographic primitives 41
2.7 Performance notes 42
2.7.1 Retrospection 44

3 The SDSI Server 47

3.1 The `Update` protocol 48
3.2 Operation notes 51
3.3 Performance notes 52

4 Supporting tools 56

4.1 The PGP-to-SDSI converter 57
4.2 The cache cleaner and locality rebuilder 59
4.2.1 Known bugs 60
4.3 The SDSI shell 61
4.3.1 Command line mode 61
4.3.2 Graphical user interface mode 62
4.3.3 The code, adding new commands 66

A Building and Porting SDSI 70

A.1 The SDSI source tree 70
A.2 How to build the distribution 71
A.3 How to port the distribution 72
A.3.1 The platform-dependent header files 73
A.3.2 The cryptography routines 75

List of Figures

2-1 The `sdsiSimpleString` data structure. 12

2-2 A certificate that demonstrates the idea of current principal. 16

2-3 An example principal definition. 19

2-4 An object-building example. 33

2-5 A `Principal:` with three servers. 36

2-6 The `Cert:` used to performance-test the library. 44

3-1 An `Update.Order` that adds a new certificate. 50

3-2 An `Update.Order` that revokes a signature. 50

3-3 The thirty-ninth certificate of a principal with 100. 54

4-1 An example of `pgp-to-sdsi` converting a PGP public key ring. . . . 58

4-2 The list of `sdsish` commands. 63

4-3 Examples of creating and displaying variables in `sdsish`. 64

4-4 An example of using the `sign` command in `sdsish`. 65

4-5 The `sdsish` command declaration for the `get` command. 67

4-6 The `sdsish` command function `do_get` for the `get` command. 69

A-1 The glue in the header file `sha/sha.h`. 78

List of Tables

4.1 `sdsish` command definition argument types. 68

Chapter 1

Introduction

The Simple Distributed Security Infrastructure[10] is a simple public-key infrastructure being developed by Professors Rivest and Lampson, in response to the complexity of proposed infrastructures like X.509[4].

In SDSI, a public key that is used to sign statements is called a *principal.* Any principal can be a certificate authority, signing certificates and other objects that are then distributed from the principal's one or more online Internet servers.

Most importantly, a principal can issue certificates that bind names locally significant to it to other values. Together, these certificates define the principal's local name space. When a local name is bound to another principal, also with its own name space, the two namespaces are said to be *linked.* Linking namespaces in this way allows indirect naming, and gives a public-key infrastructure that is based on the idea of "six degrees of separation"[1].

SDSI also features very flexible and readable group definitions, that can be used to write very auditable access-control lists for objects served from SDSI servers.

A library has been designed and written in C that implements the SDSI Version 1.0 specification. An object server, and other useful tools, have also been written in C and are based on the library.

[1]Thanks to Chris Tarr for making this connection. This is the idea that, if you have 20 friends, and each of them has another 20 friends, and so on four more times, you can "reach" anyone in the world.

1.1 The SDSI library

The SDSI library is written in C and provides a collection of functions that allow application programs to use SDSI. Chapter 2 includes sections describing the more complicated designs in the library, including the error handling mechanism, the memory manager, and the cache system.

1.2 The SDSI server

The SDSI object server is also written in C and makes heavy use of the SDSI library. While it acts as a server only for the "core" SDSI protocols (`Get`, `Reconfirm`, `Membership`, and the added `Update`), it could be expanded to serve other, user-defined SDSI protocols, or be used as the basis for another, separate server program. The SDSI server is covered in Chapter 3.

1.3 Supporting tools

Some other support programs have been written to enable users to make good use of SDSI, and they are discussed in Chapter 4. The most important of these tools is a command-line shell that can be used to perform SDSI queries, sign and issue certificates, and that also serves as the anchor point for the SDSI user interface[1]. Other tools include a utility that converts PGP[2] public key rings to signable SDSI certificates, and PGP private key rings to SDSI principal definitions, and a utility that rebuilds and cleans up structures based on the SDSI object cache.

Chapter 2

The SDSI Library

The SDSI library is the heart of this SDSI implementation; the SDSI server and all of the other SDSI tools are based on it. As expected, it provides functions that embody "high-level" SDSI ideas, like executing a `Get.Query`, signing an object, and checking an object's access-control list. But the library also has sets of "low-level" functions, and accompanying disciplines, that callers need to be aware of. A low-level side of the library was necessary to give real form to the high-level SDSI ideas, a difficult task when working in a practical, basic language like C.

This difficulty was a result of the fact that, even though SDSI is promoted as a *simple* infrastructure, it gains much of its simplicity from its *flexibility* - and ultimate flexibility is difficult to program for in a basic language. The main SDSI data structure, the S-expression, is a human-readable, LISP-like list that can be of any size. This representation, combined with a loose object syntax and flexible semantics, means that SDSI objects tend to be large, complex, and actually dependent on one another in many different ways for their full significance and meaning.

This chapter attempts to explain the problems that this flexibility caused while designing the library, and the design decisions that were made to cope with it.

The first problem this implementation faced was that C is not as well-suited to juggling a large number of big, complex data structures as it is to juggling a large number of automatic variables. So, the SDSI library has a memory manager that takes care of allocating SDSI objects, and its attentive use removes from the user the burden of worrying about when those objects need to be destroyed.

SDSI's flexibility also means that most functions make heavy use of other functions within the library, with some making good use of recursion, too. Because of this, being able to effectively deal with errors at any point during execution was another problem that led to the creation of the library's error-handling facilities, which interact a great deal with the memory manager.

To keep SDSI network queries to a minimum, the library maintains an object cache. Searching this object cache as efficiently as possible - especially on large servers - while continuing to support SDSI's allowance of detached signatures was another challenge. The idea of building cache subsets based on locality grew out of this.

Finally, the network protocol functions, the cryptography routines, and the functions that evaluate and manipulate SDSI objects needed to be general and expandable, to make for readable code and to provide for future needs, so parts of their constructions are unusual and deserve discussion.

The designs of these parts of the library will be covered in the remainder of this chapter. The general framework that the library functions work in is introduced first, in Section 2.1. Section 2.2 discusses the memory manager and error handling facilities, the cache manager and localities are described in Section 2.3, and object manipulation is covered in Section 2.4. Section 2.5 is about the network protocol support functions, and, finally, how the cryptographic side of the library works and is built is discussed in Section 2.6.

However, these sections, which dive deep into the decisions and ideas contained in the most important parts of the library, do not cover all of the library's code. Time constraints prevent giving full attention to all of the more than 10,000 lines that make up the library. The SDSI library documentation[8] includes a documented prototype and description of each function, and, since one of the chief goals was to make the library code[7] well-commented and readable, code fragments usually follow straight from the English comments that precede them. This makes the library itself present a good picture of the design of its remainder.

```
/* sdsiSimpleString */
typedef struct sdsi_simplestring {
  long int length;
  long int allocatedLength;
  int word_hint;
  _SDSI_TYPE_BYTE *string;
} sdsiSimpleString;
```

Figure 2-1: The `sdsiSimpleString` data structure.

2.1 General library framework

To support the development of a large library, a comfortable foundation is always needed. The most important parts of this "library framework" are the most common data structures, used everywhere by the library. Other heavily used features of this library's framework are the central *library context*, the SDSI reserved words functions and error codes, and other abstractions.

2.1.1 Fundamental data structures

SDSI's S-expressions encourage several basic data structures, and a library usually has other structures involved in maintaining additional abstractions. The design of these structures is crucial because they see so much use, and they tend to reveal interesting design choices; for example, the C data structure in this library that holds a SDSI octet-string is called a `sdsiSimpleString`, and is shown in Figure 2-1.

Within this data structure, `string` is a pointer to a buffer of size `allocatedLength` that has been allocated to hold the contents of octet-string itself. The actual length of those contents in the buffer given by `length`.

The curious `word_hint`, when not `-1`, is a hint used by the reserved word testing functions `sdsi_what_word` and `sdsi_is_word` to speed the process of determining if a sdsiSimpleString holds a SDSI reserved word. (See Section 2.1.4 for an explanation of SDSI reserved words.)

The remainder of this library's common data structures will not be presented in such detail here, since their actual definitions can be found in the code[7]. Instead, just the name of each abstraction, and English descriptions of its purpose and important contents are given. This is done to provide some justification for the structures used by this library, and guidance for similar decisions in other implementations. Again, this is only a list of the library's most common data structures; others, like the protocol state structure, are more specific to certain sections of the library, and will be introduced elsewhere.

- The `sdsiSimpleString`, used to hold the smallest unit of SDSI object data, the octet-string, has already been introduced.

- A `sdsiString` represents a SDSI string. A complete SDSI string is not just an octet-string: it is an octet string that can be accompanied by a presentation hint. A string has one pointer that points to the octet-string that is the "real" string, and another that can point to a presentation hint (which is actually stored as another octet-string).

- A `sdsiList` represents a list of SDSI S-expressions. Each instance of this data structure works much like a list in LISP or Scheme - there is a pointer to the head element of the list, and a pointer to the remaining sublist. This list representation reminds one of LISP's `car` and `cdr` operations, and makes recursing or iterating on lists very easy.

- A `sdsiObject` is the actual S-expression structure, and is the main data type used in the library. Each SDSI S-expression is either a list of S-expressions, or a string. Accordingly, there are two pointer fields: one pointer will point to a list, or the other will point to a string. One other field is a pointer to the S-expression that is the object's *current principal.* The notion of current principal is fully explained in Section 2.1.3.

- The SDSI library has abstractions for character stream sources and sinks, which are kept in `sdsiInputStream` and `sdsiOutputStream` data structures, respectively. These structures holds a pointer to a user-selected function that reads or writes the characters, and additional state for the stream. Stream abstractions are extremely valuable, and allow the object scanning and printing routines to easily work with terminals, disk files, or network connections.

- A `sdsiTime` structure holds a time value. Normally, the contents designate a particular time, stored as a number of seconds and microseconds since midnight UTC, January 1, 1970. Occasionally, this will just contain values representing the length of a period. Time structures can also be chained together into linked lists, which allows them to be passed "in parallel" with a list of S-expressions.

2.1.2 The library context

While performing some high-level SDSI operation, there are certain implied values that will be significant throughout execution. For example, the public and private keys of the principal currently in use by the caller of the library represent the *current speaker* using the library. These keys are used whenever a query needs to be signed, or an older signature checked - actions that might happen many times during the course of executing one high-level function call.

These keys, and other values important to "coloring" an interaction with the library, could be passed in to every function, and threaded through a call chain. However, explicit passing makes it difficult to add new things to the set of threaded values, adds to function call overhead, and doesn't look good.

The usual solution to this dilemma is to introduce some global variables. This SDSI implementation does just that, collecting all user-supplied and library-internal global items into a global data structure, called the *library context.* This context is not directly manipulable except by library functions themselves; library callers must use provided functions to manipulate the user-supplied parts of this state. The library context contains:

- A pointer to the previous context. It is somewhat possible to *reenter* the library, say, in a signal (interrupt) handler, and perform some SDSI operations. There is a function to "enter" a new library context, and one to exit the current context;

these push and pop this stack of contexts. Since this context is global and mutable, if the library is to ever be reentered, this ability is needed.

- The SDSI objects that are the current speaking principal's `Principal:` object, his signing `Private-Key:`, and his decryption key (which is usually the same as the signing key). If this speaking principal is acting as a proxy for another principal (as a server does, when answering a query "as" one of its clients), this other, client, principal is also stored here. These global items are frequently referenced.

- An object in which all credentials used to complete a library call will be collected can be put into the library context. This is especially useful during a proxy signing operation, to gather all credentials that identify the signer as a member of the `Group:` of a given `Delegation.Cert:`. Throughout the arbitrarily complex group-membership check, all credentials that were needed to satisfy the check can be collected so that they may be included, with the `Delegation.Cert:`, in the signature, relieving the need for the signature code to separately discover the required set of credentials.

- A list of the speaking principal's *local clients* is stored here, as a hint. When present, it identifies those principals that the speaking principal serves for - i.e., any principals that this principal would be willing to answer queries directed to. This is kept as a global hint as a performance gain for servers, who very often need to check principals against their client lists.

- This context's current *execution frame*. This is a number that corresponds roughly to the current depth of the library call stack, and is used by the error handling facilities and memory manager to correctly scope heap objects. See Section 2.2 for more information on memory management and error handling.

- A pointer to a set of *continuation* buffers, and a pointer to the member of that set currently in use. A continuation captures a point of execution; these

```
( Cert:
  ( Local-Name: head-nurse )
  ( Value: ( ref: Mass-General head-nurse ) )
  ( Description: "The current head nurse at Mass General Hospital." )
  ( Signed:
    ( Signer: ( Principal: ( Public-Key:
                             ( RSA-with-SHA1:
                               ( N: =Gt802Tbz9HKm067= )
                               ( E: #11 ) ) ) ) )
    ... ) )
```

Figure 2-2: A certificate that demonstrates the idea of current principal.

continuation buffers are the most important part of the error handling facilities, discussed in Section 2.2.

- Two structures, that hold the global state of the heap and cache managers.

2.1.3 Objects' current principals

A slightly nonobvious but important SDSI idea is that of an object's *current principal.* This can be thought of as an attribute of every object and subobject - right down to strings - that indicates the principal that "spoke" it. Normally, when looking at an entire, top-level SDSI object, it is obvious to a human viewer how the different parts of the object are signed, and hence "spoken", by different principals. But this essential information is lost when program functions descend into subobjects unless those subobjects are explicitly tagged with it.

For a simple example, see the certificate in Figure 2-2. When viewed as a whole, it is obvious that `( ref: Mass-General head-nurse )` was spoken by the `Principal:` given in the `Signed:`, and that this principal is the one that needs to be contacted first when resolving the `ref:` (to get a value for `Mass-General`).

But, when nested function calls start working on subobjects of this certificate, if one of them is given the `ref:` subobject and needs to resolve it, where does this initial, or *current*, principal information come from? Unless it has somehow been passed down through the function calls to the `ref:`-resolver, the resolution cannot proceed.

In this implementation, before any SDSI object that is received over the network or read out of the cache is handled, it has its current principal "set" throughout. What this means is that the object is descended into, and each subobject, right down to the individual strings, is tagged with a pointer to the SDSI `Principal:` object that spoke it, according to the following loosely stated rules:

- If this subobject is a `Signed:`, the principal named in its `Signer:` field is this subobject's current principal.
- If this subobject has an appendix mode signature, the principal named in its `Signer:` field is this subobject's current principal.
- Otherwise, this subobject's current principal is the same as its parent object's current principal.

Because this is done, all instances of SDSI objects inside the library carry this extra information of who spoke them, which is invaluable where the need for such information is basically implied - as is done in the convenient, simple SDSI constructions like `ref:`s and `Group:` definitions.

2.1.4 Reserved words

The SDSI specification introduces the core SDSI object types, and in doing so implies a set of reserved words, like `Cert:`, `Group:`, and `ref:`. These reserved words are necessarily referred to a great deal throughout any library's implementation.

To prevent having to type the actual text of a reserved word each time it is needed in some octet-string (and in the process, risk misspelling it), and to aid in determining which, if any, reserved word a given octet-string already holds, this library implementation has a numerically indexed set of reserved word octet-strings, and a group of functions that map back and forth between indices and equivalent octet-strings.

The reserved word set is built automatically during the library build process, and the values that index the set are available as C preprocessor symbols. This way, when a reserved word octet-string is needed or tested, a preprocessor symbol is used instead of a literal string value - which means that typing mistakes will be caught by the compiler as "unknown symbol" errors.

A small optimization in the reserved word system consists of a "hint" that is left behind in the data structure of an octet-string found to contain an indexed reserved word: the index value is left there as a first guess to speed future calls to map the octet-string to a reserved word index value.

2.1.5 Error codes

Many possible errors can be encountered during the execution of a SDSI operation. These errors, and the no-error case, are distinguished by different *error codes.*

In this implementation, most functions return an error code as their value. The last called function's error code is also always returned in a global variable, which is available to both the library and its external callers.

Similar to the reserved word index set, these error codes are also indexes into an array of English strings that attempt to explain the error. This table of strings is built automatically during the library build process, and the error codes that index the set are available as C preprocessor symbols. Whenever the library encounters an error condition, or the library or a caller need to check for a previous error condition, the current error code becomes or is checked against one of these symbols.

2.1.6 Principal definitions

A SDSI principal is fully defined by a pair of S-expressions: the principal's `Principal:` object, which contains his public key, and the corresponding `Private-Key:` object. These two items need to be stored and accessed together, and are called a *principal definition* by this library.

Principal definitions are stored in files, and one is read into the library when

```
( Principal-Definition:
  ( Public:
    ( ( Principal: ( Public-Key: ( RSA-with-SHA1:
                                   ( N: #B8D1C7A2 ... )
                                   ( E: #11 ) ) )
         ( Server-At: oak.lcs.mit.edu spruce.lcs.mit.edu ) ) ) )
  ( Private:
    ( Encrypted:
      ( Key-Hash: ( SHA1 #A2AA889BB4CC35A3A9563C20D4FAD411A8C38887 ) )
      ( Ciphertext:
        #3B843578F73ED7B65FECCDC1DF1 ... ) ) ) )
```

Figure 2-3: An example principal definition.

directed to become the current "speaking principal" in the library context. Because these definitions are stored on disk, and have a sensitive component (the private key), definitions are split into a public section, which is stored as plaintext, and a private section, which is stored encrypted. Currently, the only members of these two sections are the `Principal:` and `Private-Key:` objects, respectively.

The private section is encrypted, or *sealed*, in a passphrase, much like private keyring data is sealed in PGP[2]. A passphrase is turned into an IDEA[5] key using MD5[9], which is then used by the library to unseal the private section of the definition.

Figure 2-3 shows an example of a principal definition as it is stored on disk. Every principal in SDSI, whether it represents a human, a server, or something else, has a principal definition. When starting the SDSI server, or the SDSI shell or user interface, one of these principals is loaded by the library. The actual file containing the principal definition to be used can be specified in a library call, or by values contained in runtime environment variables.

2.1.7 Files

Occasionally, a SDSI library or application needs to be able to find files or directories in permanent storage. For example, a call to load a principal definition does not always specify the explicit pathname to the principal definition file, nor should it have to - a user has his favorite principal definition file somewhere, and he usually means *that* one. Similarly, cache search function calls should not have to specify the location of the cache directory - it should also be implied some way.

In traditional Unix applications and libraries, the pathname of a needed filesystem element that has not been specified for the operation in progress, but that can be implied, is implied in one of two ways:

1. Runtime environment variables can be present that yield the pathname.
2. A fallback pathname that has been hard-coded into the application or library can be used.

This SDSI library follows this model. For example, the favorite directories of the library, like the object cache and localities directories (described in Section 2.3), can be named in specific environment variables (`SDSI_CACHE_DIR` and `SDSI_LOCALITIES_DIR`, actually). If they are not present, these directories are searched for under the directory named in any current value of the environment variable `SDSI_DIR`, then under the user's home directory. As a last resort, the library will search for them under a hard-coded, Unix-centric `/var/sdsi`.

Other SDSI implementations should follow this model. The ability of a single environment like `SDSI_DIR` to locate all SDSI filesystem items a library or application needs, with per-item overrides as needed, is very useful.

2.2 Memory management and error handling

Any SDSI implementation will depend on good complex object support and good error handling, provided either by the programming language, or by the library itself, for the following reasons:

1. SDSI objects tend to be complex and big, so space for them usually has to be allocated out of dynamic (heap) memory.

2. Whenever space for something is allocated from a heap, that space must be freed as soon as that something cannot be referenced again - otherwise, the resulting "memory leaks" will eventually reduce the free heap space to zero.

3. Most SDSI library functions will return and accept SDSI objects, and place many library calls themselves. This means that most function bodies will deal with a lot of objects during the course of their execution.

4. Because SDSI library functions do call other functions so routinely, long call chains are common.

The third and fourth reasons are the most important of all. The sheer number of objects that most functions traffic in means a large number of objects can potentially go "out of scope", or become unreferenceable, during execution, and must be freed to prevent memory leaks. Long call chains makes the ability to gracefully handle "deep" unrecoverable errors essential.

Some languages address these issues for the programmer (Scheme and C++ come to mind). However, native C provides the programmer no unconscious mechanisms for deallocating objects that have gone out-of-scope, or for handling sensitive data (that needs to be zeroed out when appropriate), or for aborting a call chain, and this was recognized early in the design of this library.

Given this, there were several possible solutions. For memory management, either the function bodies could explicitly deallocate objects just before they fell out-of-scope, or an unconscious mechanism for deallocating objects that are out-of-scope could be developed. The latter, the development of a garbage collector, was chosen for three reasons:

1. Explicit deallocation calls would clutter the library code, and at times even cloud a code fragment's intent. These calls are unnecessary with a garbage collector.

2. Every time a subtle scoping change is made, or a new function call is added, all explicit deallocation calls have to be checked for completeness and correctness.

3. Garbage collection can handle unusual program flow control (as will be introduced with the error handling facilities), but explicit deallocation calls can be missed if unexpected non-local jumps can happen.

Strangely similar to the deallocation choices, there were two possibilities for comprehensive error handling. Either every function call within the library could be wrapped with error-checking code that would explicitly `return` and propagate errors back up a call chain, terminating it, or a more unconscious mechanism, à la C++, where errors are "thrown" automatically to the top of a call chain and "caught", could be created.

It was chosen to build such an error throwing mechanism, because all of the explicit error-checking tests and `returns` would have cluttered the library code even more than deallocation calls would have. Not having to worry about checking for error reports from a code fragment's function calls makes those fragments much more readable, without sacrificing the ability to handle those errors *somewhere.*

These two decisions, to develop a garbage collecting memory manager and a catch-and-throw error handling mechanism, resulted in two systems that are very intertwined. Both are involved in scopes and flow control - the garbage collector needs information about which functions have exited or have been aborted to determine which objects on the heap are garbage, and this is information that only the catch-and-throw code can provide.

These remaining sections discuss these two systems in more detail.

2.2.1 Global state

The memory manager and error mechanism both keep global state, some of it shared, in the library context:

- The memory manager keeps in the library context a pointer to the head of a linked list of all of the individual blocks that have been allocated on the heap. A block is a single buffer of storage.

- In the library context, a value is stored that represents the context's current *execution frame*. The memory manager uses this value to represent the current allocation scope, and all newly allocated objects are made members of this scope. This value is updated by the catch-and-throw error mechanism, whenever a library function begins execution, finishes, or throws an error. This value can also be artificially incremented and decremented by the library, or by applications, wherever new allocation scopes are desired.

- The error mechanism keeps a set of *continuation buffers* in the library context. Saving a continuation means capturing the current point of execution, and calling a saved continuation means resuming execution exactly from that captured point - the entire machine state is restored from the information stored in the continuation. These continuation buffers make possible the non-local jumps that throwing an error requires. For each active "catch" point, one of these buffers is filled, so a thrown error will cause execution to resume at that catch point.

2.2.2 Prologue and epilogue macros

Since both are so concerned with function entry and exit, the memory manager and error mechanism each have code that must be executed at every function entry and exit point. On entry, the heap manager has to start a new allocation scope, and the error handling mechanism must save the current continuation, if this call is being made from a catch point. On exit, the heap manager has to close the allocation scope, and the error handling mechanism must release a saved continuation, if the exit point is also a catch point.

Macros called the *prologue* and *epilogue* macros exist for this, and they are used at all function entry and exit points. The use of macros makes it possible to change the necessary prologue and epilogue code in one central place, without having to actually change code in every function.

While these macros are used whenever a function begins or ends execution, they do not impose a significant speed penalty on the library. The extra work involved in calling and receiving control back from a library function is normally no more than eight simple operations, made up of just increments, decrements and assignments of memory locations. Only when a call into the library is made from the application, is the additional work of saving of the current continuation performed.

2.2.3 Memory management

Many points about how the memory manager is implemented, and how dynamic memory actually *works* within the SDSI library are important - they indicate how applications using the library should operate when it comes to managing objects that live on the heap.

These points go into great detail about how the heap is operated. They do not discuss the decision-making process that yielded these details, however, as that process consisted mostly of trial-and-error until something that worked was obtained.

Definitions of much of the terminology used below can be found in [6].

- A *heap block* is an individually allocated buffer, meant to hold a single C structure.

- Heap blocks are allocated one at a time, and a sorted linked list of all currently allocated blocks is kept, with the most recently allocated block at the *front* of the list.

- A *heap object* spans many heap blocks. For a simple example, if two C structures are on the heap (in two distinct heap blocks) and the first includes a pointer to the second, the two blocks together make a heap object. The first block is called the *parent* heap block, and the second is its only *child* heap block.

  Information about a heap object is found in its *heap object description*, which is kept with the parent block. Part of this description is the set of locations within the parent block of its pointers to its child blocks. This provides the *reachability information* that the memory manager needs in order to walk complex objects on the heap during garbage collection. Every SDSI C data structure that lives on the heap and has pointers to other C data structures on the heap, makes a heap object, and so has a corresponding heap object description.

- Every heap block is a member of some allocation *scope*, or *frame*. Frames are nested inside each other. Once the frame that a heap block is a member of has closed, the block becomes garbage and is eligible to be collected.

- When comparing two possible frames, the *least*, or *better* one, is the one that contains the other in the nesting order - in other words, the better of two frames is the one that opened first, and will close *last*.

- When a new heap block is first allocated, it is a member of the allocation frame that is current. The current allocation frame corresponds roughly to the library function currently being executed.

- A heap block can be *promoted* to the frame enclosing the current allocation frame in the nesting order, thus preventing it from being made eligible for collection when the current allocation frame closes.

- Alternately, if a heap block is a member of a heap object, and one of its ancestor blocks is a member of a frame that encloses the current allocation frame, the former block will also be spared from garbage collection when the current allocation frame closes - it *inherits* membership in its ancestor's frame.

  So, block promotion, plus the inheritance feature, allows heap objects to live beyond a soon-to-close frame. This behavior is used by library functions to return S-expressions to their callers that will not be immediately eligible for garbage collection, without having to explicitly promote every block in the heap object.

- For efficiency, the following discipline is used to great advantage in the SDSI library: in general, any heap-based object that is passed as an argument to or returned as a value from a library function is always considered *read-only*.

This allows for the sharing of large subobjects on the heap between completely different objects at use within the library. The resulting savings in duplication time and heap space from the use of this discipline is enormous.

- When a shared heap-based object does need to be modified, it can be *locked*, which sufficiently duplicates the object for modification by only copying the *top* of the object. The top of a heap-based object is defined as the parent block of the object, plus the tops of any children blocks also designated to be locked in the parent's object description[1].

- Heap blocks that are allocated or promoted to the outermost enclosing frame are never eligible for garbage collection, since this best frame (frame zero) never closes. These blocks, and any heap-based objects these blocks are the parents of, are called *immortal* objects within the library, because they never die.

- Part of a heap object description is a pointer to an optional function that will be called right before the parent heap block is destroyed by the garbage collector. This can be used to release other, non-heap resources contained in the heap block about to be destroyed.

- The order in which the parent and children of a heap object are destroyed by the garbage collector is unspecified.

- Every block in the allocated blocks list always indicates the *least* allocation frame that was *not* closed the last time the block was at the head of the blocks list - this is called the *greatest living frame* at that point in the blocks list.

- All of the greatest living frame values on the blocks list are interpreted as a monotonically decreasing lower bound on the allocation frame that, for the remainder of the blocks list, has not closed yet.

- The garbage collector itself is technically characterized as a *mostly-precise mark-and-sweep* garbage collector[6]. To be mostly-precise means a garbage collector

[1]The need for the idea of an object's top is best seen by the top-description of a `sdsiList`.

> ...permits a mixture of precisely and conservatively identified values. For example, values in registers and on the stack might be treated conservatively, but objects on the heap would be fully described and precisely scanned.

This describes the SDSI garbage collector very well. The pointers that may be in register values and stack locations tied to allocation frames that have not closed are ignored (and are thus treated conservatively), while the garbage collector relies completely on its knowledge of which allocation frames *have* closed, and the exact descriptions it has of the objects it finds that were associated with it, to collect dead blocks.

The garbage collector marks all blocks that are members, no matter how indirectly through parents, of allocation frames that have not closed. Then, it sweeps over all blocks again, destroying and releasing all blocks that have not been so marked.

2.2.4 Error catching and throwing

The use of the error handling mechanism mostly centers around deciding where errors will be caught, and where they will be thrown. The latter decisions are easy to make - any time a library function encounters an unexpected condition that forbids sensible continuation of its execution, it throws an error.

A macro is provided to make this easy: it takes a SDSI error code, and a `printf` argument body (which displays any provided message, if library debugging is turned on), and throws the error back to the most recent catch point.

The former decisions are somewhat more difficult to make. Normally, when a library function calls another library function, and it encounters an error, that error also prevents the sensible continuation of the first function's execution. So, most library calls never catch errors from the function calls they make.

Occasionally, however, a library function does have strategies for continuing when something it tries to do fails. When this is the case, there is a macro that it can wrap an individual function call in, to catch any errors it signals. This macro is based on two functions that can be used to explicitly turn "downstream" error catching on and off within a library function.

These functions manipulate the set of error continuations in the library context, using one for every active catch point.

The error code of the last error thrown is always left in a global variable, available to the library and to callers.

2.3 Cache management

An S-expression cache is useful when working with SDSI, for storing certificates and other SDSI objects generated locally, along with objects received as responses to network queries. A cache prevents repeated queries of network servers, which are assumed to be busy enough already, and, in any case, gives faster response time than a network query could.

The general functionality that a SDSI object cache manager must provide is:

- A cache that is kept in permanent storage.
- Functions to add or remove an object from the cache.
- A search capability, that, given a SDSI template, finds all objects and subobjects in the cache that match it.

This last piece of functionality, the search behavior, is necessary to tolerate all of the flexibility that SDSI objects can exhibit. If a search is executed looking for all `Cert:` objects, matching objects might be found quite literally *anywhere* inside the objects in a cache - since SDSI places very few limits on where objects of given types can appear.

Another good piece of functionality a library's cache manager could provide is a cache kept in *temporary* storage, in memory. This cache would be considered part of the regular permanent cache for the purposes of searching, but its elements would not live beyond the process' lifetime. This allows objects that only need to be found in cache searches for a short time to avoid being stored on disk.

All of this functionality is provided in this implementation's cache manager. Objects in the permanent cache are stored as files on disk, and the temporary cache is a linked list of objects kept on the heap (made *immortal* so they would never be destroyed by the garbage collector). A recursive search into all of these objects with the given template is done to satisfy a search requests.

However, once the library was relatively finished, and testing began, it became clear very quickly that this brute-force search over every last piece of data in the cache was inefficient. This was especially obvious when the cache was searched for objects *and* their signatures. One full pass over the cache was needed to pick up the objects, followed by a second full pass to pick up the relevant signatures. Even worse, if a signature had reached the end of its validity period, a *third* full pass over the cache was needed to check for a reconfirming signature.

On a server, all of these full passes over a modestly sized cache would drive query service time into the realm of a minute or more, and this was certainly unacceptable. A solution was needed.

2.3.1 Localities

The solution came with the realization that most cache searches are executed to look for objects with a specific *locality* in mind. Locality is (perhaps somewhat poorly by Webster's Seventh Dictionary) defined as "a particular spot, situation, or location." The following concepts of locality are not specific to this implementation at all - any SDSI library design should make some use of these ideas.

For example, when searching a cache for objects to satisfy a particular `Get.Query`, besides the search template given in the `Template:` field of the query, there is another piece of information relevant to the search: the only objects in the cache of real interest are *those that have been signed by the principal named in the* `To:` *field of the query.*

This means that the search can be limited to a subset of the objects in the cache - the subset defined by those objects that share the locality of having been signed by that particular principal. If this set of objects can be found quickly, only they would have to be matched against the template, and only at their top level, since unsigned inner objects, even if they did match the template, aren't of interest.

The `Get.Query` example identified one class of locality within a SDSI object cache: all objects signed by a particular principal P form a locality. In fact, all objects signed by other principals P' also form localities. All of these localities are of a certain type: they are all *speaker* localities - a locality that is the subset of all cache elements signed by a given speaker.

Speaker localities work very well for limiting cache searches to those objects that are of interest to a `Get.Query`, but they are not necessarily useful for all cache searches. Other locality types emerge from other searching patterns that are used on the cache:

- An *object* locality is the collection of signatures on a specific SDSI object. This is used to get the list of signers of a particular object.
- A *key* locality is the collection of all keys of a particular high-level type (public, private, and shared-secret). These localities are extremely useful for finding the key that matches a particular `Key-Hash:`.
- `Signed:` localities are very specific to signature reconfirmation. One of these localities consists of a particular `Signed:` and all of the other `Signed:`s that name that same `Object-Hash:`, plus all of the `Signed:`s that sign any of *them*. This gives the signature reconfirmation code quick access to the signature objects that could be either new signatures on an original object, or signatures *on* a signature on the original object.

This SDSI library's cache manager can have locality support compiled into it. When a locality applies to an impending cache search, to be made by the library itself or by an application, the cache manager can be advised of it. (But, because actual locality support *is* an optional part of the cache manager, it should not be assumed that objects returned by a locality-advised search necessarily satisfy the requested locality.)

The actual localities take the form of separate files, one per specific locality, that contain all of the objects from the cache that make up the applicable subset. For maximum speed, the most recently used locality files are themselves cached in memory by the cache manager. Since localities can be large, this memory caching leads to a process-size issue, especially on servers, and this is discussed in Section 3.3.

Additionally, all localities, of all types, can only be built by running an external program, `sdsi-rebuild-cleanup`. This is one of the support tools discussed in Chapter 4. Building localities is a rather consuming task, which is why an external program, and not the library, does it - the library only uses them. How often localities are rebuilt is left to the discretion of any user or server administrator that has a SDSI cache to manage, and wants to use localities to speed searches up.

However, because localities are not built (or even updated) by the library, this means that new objects placed in the cache by the library will not be made part of the localities until the next rebuild. Since localities exist to remove the need to search the actual objects of the cache, when they are not complete, how the library should treat the cache objects not represented in them is another question.

To compensate for this, this library (as should any other implementation that uses localities, but doesn't constantly update them) allows the application to specify one of three cache searching behaviors, which apply *only* when there are incomplete localities (i.e., when there are objects in the cache that are younger than the localities, and therefore cannot be represented in them):

- The *coarse* search behavior forces the cache manager to completely ignore cache objects not represented in the localities. Cache searches are done exclusively using the localities.

- The *normal* search behavior searches the localities as normal, and performs a full sweep of only those cache objects not represented in the localities. This can miss some new detached-signature associations, but will pick up most new relevant objects, at a slight speed cost, until the localities are next rebuilt.

- The *exhaustive* search behavior ignores localities completely, and performs full cache sweeps as if there were no localities. This always guarantees that all detached-signature associations will be correctly made, although it imposes a significant speed penalty because of the full cache sweeps.

These choices are extremely relevant to the SDSI server program developed from this implementation. The decision of how often to rebuild localities from a server's cache, combined with the choice of its incomplete-locality search behavior, determine a server's performance. Notes on operating one of these SDSI servers can be found in Chapter 3.

2.4 Object evaluation and manipulation

The S-expression data structure will be the most common data structure in any SDSI library. In some sense, SDSI libraries and applications do nothing except understand existing S-expressions, and build new ones[2]. A little more specifically, the most basic object manipulations identified as typical are:

- Given a SDSI object, find the single subobject that is the value of a particular attribute within the object. An example of this is finding the value of the `Date:` attribute of a `Signed:` object. This is the dominant operation a library needs to perform on finished objects.

- Attempt to match a template to a SDSI object. This directly refers to what needs to be done to match a `Get.Query`'s `Template:` template to candidate signed objects.

- Evaluate a SDSI object to its truer "meaning", by removing all wrapping constructs and by transforming or resolving it as necessary. Examples of SDSI wrapping constructs are the wrapped-object `Signed:` form, and

[2]This is similar in feel to how addition takes two numbers and makes a third. The idea of *everything* being information passing through filters and coming out as new information is fascinating - complexity can be found in information, or its filters. SDSI puts complexity its information, which is as it should be - addition does the same, and addition is not only *very* powerful (thanks to the large set of numbers you can make to use with it), but it is also secure.

```
/* construct a template that looks like:
   ( Delegation-Cert:
     ( Template: ( ,messagetype ) )
     ( Group: ) )
*/
ADD_OBJ(obj1 = ADD(NULL, SDSI_WORD_TEMPLATE), ADD_SS(NULL, messagetype));
ADD_OBJ(template = ADD(NULL, SDSI_WORD_DELEGATION_CERT), obj1);
ADD_OBJ(template, ADD(NULL, SDSI_WORD_GROUP));
```

Figure 2-4: An object-building example.

Assert-Hash:. Object types that can be transformed are the Encrypted:, and Encrypted-Macro: types, and ref: objects are resolved.

- Add an octet-string, a list, or another object to a SDSI object, so that new objects can be built for transmission or storage.

Without this functionality squarely in place with an easy-to-understand interface, any SDSI code would be difficult to read and debug.

Evaluating SDSI objects is, at this topmost level, the simplest task in this group. Whenever an object needs to be evaluated, if it is a wrapped-object Signed: or an object wrapped in an Assert-Hash:, the wrapping is removed. If it is an encrypted object, it is decrypted, and if it is a ref:, it is resolved into its value. As many of these transformations as possible are made, until the object has been fully evaluated.

Next, it is very important that the functions that add to SDSI objects are easy to use. An intuitive interface for these functions in this library was eventually developed; each of the functions that adds an octet-string, a list, or an object to a destination object, takes the item to be added and a pointer to the destination object. If no destination object is provided, a new one is started. The item is then added to the destination object, and the destination object is returned as the function's value.

The utility and importance of this specific interface is better seen in an example of its use. Creating the same object that the code fragment in Figure 2-4 does, by instead referencing data structure members and allocating memory explicitly, would be a great jumble of function calls and C indirection.

Using the octet-string stored in `messagetype`, this code completely constructs the complex template object described in the comment and leaves a pointer to it in `template`. (The `ADD_` symbols are preprocessor macros that simply take the place of longer function names, and the `ADD` macro adds a reserved word, represented here as a preprocessor symbol, to an object.) These object-building functions were deemed absolutely necessary for writing this library in C; functions with a similar interface should be considered in any other language without serious object support.

Another important decision in this library was to make the template matching function very general and powerful. This function does more than merely make sure that an object matches a template (although it can be used to do only that); it is normally used to *impress* a template upon an object. This returns a new object, similar to the original one passed in, but with elements ordered according to the order of the elements of the search template. This returned object is called an *impressed object.*

The real generality and power of this impressing operation is this: while the elements of impressed objects are always ordered according to the template used to impress them, they are not necessarily elements from the *original* object passed in with the template. If the original object completely fails to match the template, it is usually assumed that it is a wrapped or transformed object, and that the result of evaluating it might match the template. So, objects that a template completely "misses", but that can evaluate to a different form, are evaluated and checked against the template, repeatedly, until some matching is achieved, or until no more evaluation is possible.

This "evaluating before admitting failure" is what makes it possible to successfully impress complex templates onto objects without having to worry if they are, in whole or in part, composed of ciphertexts, `ref:`s, or other object types that are usually evaluated for their meaning. Since most of the time, templates are constructed to match and find the parts of an object's true meaning, this behavior makes good sense. However, occasionally this and other template-matching behaviors will need to be disabled, and in this implementation they can be, on a per-call basis.

Last, because it was easiest, the functions that search for a particular attribute's value within an object were simply built on top of the template impressing function. In other words, all they do is construct a simple template designed to single out the desired attribute in the provided object, and impress that template on the target object. The resulting impressed object then contains the desired attribute value.

While this was the easiest and clearest approach, and it allows those functions to leverage off of the intelligence of the template matcher, their performance is probably not as good as it could be. Since finding an attribute's value is such a common operation, and the whole template impressing mechanism is really overkill for what it needs, functions to do this should be specially written for speed.

2.5 Protocol primitives

SDSI is a very online infrastructure, within which parties are supposed to meet and exchange information using various protocols. The SDSI specification defines three standard protocols that anchor the infrastructure: the `Get`, `Reconfirm`, and `Membership` protocols. These three protocols make certificate and membership information distribution possible.

However, these protocols only yield a minimal infrastructure, and not one that can necessarily do anything interesting. Additional protocols would increase the amount of expression possible in SDSI, and the specification notes:

> 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.

The SDSI specification also says, "We envision that a separate SDSI transport protocol can be designed on top of TCP/IP. Ultimately, this would be the preferred solution." This serves to remind not only of the dominance of TCP in today's Internet,

```
( Principal:
  ( Public-Key:
    ( RSA-with-SHA1:
      ( N: #B8D1C7A2EB ... )
      ( E: #11 ) ) )
  ( Server-At: 18.52.0.220 pine.lcs.mit.edu:8002 spruce.lcs.mit.edu:sdsi ) )
```

Figure 2-5: A `Principal:` with three servers.

but also of the fact that other application-level "media", like email, can be used to execute protocols.

Altogether, then, a SDSI implementation's set of protocol primitives is very important - it should be a group of functions that are convenient and abstract, to encourage protocol development and allow for the use of different transport media for communication. This SDSI implementation includes suitable primitives that address these two concerns.

2.5.1 Transport mechanism

First and foremost, this SDSI library's natural transport mechanism is TCP. This choice immediately points out an underspecified feature of SDSI `Principal:`s - it is unclear exactly what the members of their `Server-At:` and `Principal-At:` fields can be.

Introducing a bit of terminology, the library refers to these members as *endpoints*, because they are meant identify one end of a communications channel to which protocol messages can be addressed. The specification says only that an endpoint is included "typically as a partial URL", and gives several URL examples.

While endpoints could one day be partial URLs, this library assumes now that all endpoints do nothing more than designate an Internet machine and a TCP port number that protocol connections can be made to. These endpoints take the form of an Internet hostname or IP address, followed by an optional port name or number.

For example, a `Principal:` with three SDSI servers is shown in Figure 2-5. The three `Server-At:` members illustrate some of the possibilities for endpoint names.

The port name or number is always separated from the machine name or number by a colon, and if it isn't present, `:sdsi` is assumed. Port names and machine names are resolved according to normal Unix conventions[3], to yield a complete Internet address that a TCP connection can be opened to.

Once a connection has been opened, SDSI S-expression objects that are the messages of the ensuing protocol are exchanged over it, by sending and receiving those objects' canonical representations.

2.5.2 Protocol state

In this library, the central feature of the suite of protocol functions is the *protocol state* object that they all use. This is a data structure that holds information describing a protocol in progress, and it is used largely to shield the users of these primitives from the specific details of the message exchange. The protocol state carries values describing:

- The current input and output streams associated with the protocol connection. Normally, callers never provide these; however, they can be specified if a protocol is being driven over byte streams not associated with a TCP network connection made by the library.

- The endpoints of the protocol connection. One is the *local endpoint name*, describing the local end of connection, and two are used to describe the remote end. If this protocol state is for the client side of a connection, the *remote endpoint name* holds the originally specified endpoint name of the server, and the *canonical remote endpoint name* is the endpoint name that the connection was actually made to. (There can be a difference when there is DNS indirection at work.) These two remote endpoint names are identical when the protocol state is for the server end of a connection.

[3]I.e., `gethostbyname(3)` and `getservbyname(3)` are used to resolve host and service names into IP addresses and TCP port numbers. If the service name `sdsi` cannot be resolved, 2500 is assumed by the library.

- Timeout values, specifying how long a server should wait for an incoming connection, and how long the next message from the remote endpoint should be waited for.

- Flag values, controlling how the local and remote ends are running the connection. These flags specify how the two ends are to sign and/or encrypt the messages they send, and their last-message-hash discipline.

- A list of the principals and corresponding signatures that signed the last message received. This list defines the set of *remote speakers*, or participants, in this protocol.

- Last-message-hash information for the next messages to be sent and received, and optional values that override the keys used for local or remote message encryption, or the key and lifetime information used in local signing.

2.5.3 The protocol functions

All of the SDSI library's protocol functions take a protocol state as an argument. There are functions that:

- Listen for client connections to a particular local endpoint name. This function is used by servers to wait for client requests.

- Establish a new connection to the server listening at the remote endpoint specified in the protocol state.

- Explicitly close an open connection.

- Manage the sending and receiving of messages. Once a connection has been made and accepted, two functions handle all of the messages that travel over the connection. These functions take care of encrypting, decrypting, signing, and verifying all messages as necessary.

- Construct and send an error message to the remote end of a connection.

- Establish a new connection to the *next* server of a particular remote `Principal:`. This function works like an *iterator*: aside from the protocol state, it maintains another piece of private state, so when called in a loop, it will move its caller's connection successively through the list of a remote principal's servers.

- Iterate a query-response session with the servers of a particular remote `Principal:`. This builds on top of the previous function, by sending a given query message once the next connection has been opened, receiving a response, closing the connection, and returning the response. This kind of behavior is useful for simple protocols like `Get`, which form a single query for all servers, and address it to each one in turn, stopping when one of them returns an "acceptable" response (where "acceptable" depends on the particular protocol). This function is called in a loop until a satisfactory response is returned, or until there are no more servers to query.

One last, curious function is one that is used to discover a server's `Principal:` definition. When a query addressed to a particular principal needs to be sent to one of his servers, and needs to be encrypted, it needs to be encrypted such that the receiving *server*, and not the addressed principal, can decrypt it.

This final function, given a message type, a client principal, and the endpoint of one of that principal's servers, queries all `Delegation.Cert:`s that the client principal has issued, that allow others to sign objects of the given message type. This list is then searched for a certificate that names a principal whose `Principal-At:` member matches the desired server's endpoint. This is the actual `Principal:` that is the server, and it is to this principal the original query needs to be encrypted.

2.6 Cryptography functions

SDSI's use of cryptography is the basis of what inspires trust in it, making the cryptography routines yet another important component of any implementation. However, in a certain sense, this component also stands to be the most fragile.

Today, with the increased visibility of cryptography, more and more attention is being focused on breaking hash functions and cryptosystems. This is why all of SDSI's cryptographic S-expression objects are so flexible - they always explicitly specify which hash function or cryptosystem is tied to their use. As cryptosystems and hashes fall to attacks, new ones can be brought into service in SDSI without any change to the general object syntax in use.

This does mean, though, that implementations have to be ready to accept and provide new cryptosystems as easily as possible. An ideal implementation supports all of the suitable and "unshaken" hashes, and public- and secret-key cryptosystems that are currently available.

The vision of such a "cryptographically ideal" SDSI implementation drove the design of this library's cryptographic routines. Accordingly, there is a clean split between library functions that consider general cryptographic operations and those that take real cryptographic actions. For example, the function that takes a key and plaintext, prepares for an encryption and assembles the resulting final SDSI `Encrypted:` object, is separate from the function that actually generates ciphertexts from plaintexts. The former deals with keys, plaintexts, and ciphertexts in a very general way, while only the latter treats them as participants in a specific cryptosystem.

2.6.1 Key profiles

To allow for this break between general and specific cryptographic functions, and to easily support a large number of cryptosystems, the idea of a *key profile* was created. A key profile is a data structure that contains all of the pertinent information of a key, extracted into a generic, easy-to-use object.

Every key has properties that exist somewhat independently of the bit-by-bit details of how it is actually used within its cryptosystem. Whether a key is a public, private, or shared-secret key, whether it can only be used with signatures, how many bits of data it can operate on, and whether or not it can operate on more through chaining, are all examples of attributes that more general cryptographic routines want to know when they are dealing with keys.

Key profiles are used inside the library to carry this information alongside a copy of the key's *core* and complete break down of the key's type name. For example, the public key:

```
( Public-Key: ( RSA-with-SHA1:
                ( N: #B8D1C7A2EB ... )
                ( E: #11 ) ) )
```

would be manipulated as a profile of a public key that can encrypt and verify signatures, and that can encrypt up to 512 bits (assume `#B8D1C7A2EB`...is 512 bits long) but cannot do any chaining. Also stored in the key profile would be key's type (RSA), associated signing hash function (SHA1) and the key core:

```
( RSA-with-SHA1:
  ( N: #B8D1C7A2EB ... )
  ( E: #11 ) )
```

For manipulating key cores themselves, functions are available that:

- Turn a complete SDSI `-Key:` object into a completed key profile.
- Use a passphrase to complete the key core of an unfinished profile. This is used when a shared-secret-key's secret data is provided as a passphrase by a human user, and it needs to be turned into a useful key.
- Complete the key core of an unfinished profile with random key data. This is used for generating random session keys.

These key profiles are always passed from the general to the specific cryptographic functions (also called the *cryptographic primitives*), and they direct the latter to a particular cryptosystem.

2.6.2 Cryptographic primitives

The cryptographic primitives are the functions of the SDSI library that are actually aware of the different hash functions and cryptosystems available to the library. It is in these functions that the "rubber meets the road"[4] and calls to the third-party cryptography modules are made. Aside from the primitives used for turning `-Key:` objects, passphrases, and random sources properly into key profiles, there are very few of them.

There are only two primitive hash functions: one that takes a hash function name and indicates whether or not it is supported, and another that takes a hash function name and an octet-string, and returns its raw hash under that function as another octet-string. Several more general and complex SDSI library hash functions, ignorant of the actual supported hashes, are build on top of these primitives.

The remaining four cryptographic primitives all involve running real cryptosystems. Most of these functions' bodies consist of C `switch` statements, with one `case` arm for each supported cryptosystem. The specific *type* contained in the key profile they all take as an argument chooses which arm will run. Adding more cryptosystems, then, means only adding more arms to these existing pairs of functions:

- One function encrypts a plaintext octet-string into a SDSI object containing the ciphertext in the format most suitable for the specified cryptosystem.
- Another function takes those cipher objects and recovers their plaintext.
- One function signs a plaintext octet-string and returns a SDSI object containing the signature in the format most suitable for the specified signature system.
- Another function takes those signature objects and verifies their contents.

On top of these primitives are built the top-level SDSI library cipher and signature functions.

2.7 Performance notes

This SDSI library is a concrete, working implementation that has seen some tinkering use by its developer, and some more involved use by the developer of the SDSI user

[4]Professor David Gifford introduced me to this great term.

interface[1]. As such, there are some performance notes to relate, as well as some retrospection about the how the library was written.

Concern for the library's speed came from the sluggish response times seen from the SDSI servers brought up during the user interface development. A server's speed is a function of two things:

1. The size of the object cache it must manipulate. This is the sum total of the objects it is distributing on behalf of its clients, plus all of the other credentials it has cached from previous "network encounters". The number of objects in this cache affects a query's service time.

2. The speed of important library functions. The remainder of a query's service time, once the cache size factor has been removed, lies in those functions needed to prepare the response.

An analysis of the first effect is in Section 3.3. To give some insight into the library's role in server performance, a crude program was written to test three library functions important to servers: hashing, creating signatures, and verifying signatures. This program was run on an unloaded Sun SPARCstation 5 with 28MB of memory to gauge how the library performs.

The results were less than exciting. Performing 10,000 SHA1 hashes of the simple, unsigned `Cert:` in Figure 2-6 took 14.78 seconds, showing that the library can generate around 680 hashes per second.

Signing with RSA was the slowest of the three operations. To sign the same `Cert:` 100 times with a 512-bit key took 50 seconds, meaning 2 signatures were generated per second. Signature verification was much faster - the library can verify around 30 signatures per second, verifying 100 times a signature made on the test `Cert:` in only 3.03 seconds. The sluggish signing time, thankfully, is at least one-third due to the actual RSA modular exponentiation. (Even though verification is the same operation, it ends up being much faster because the public exponent is usually *very* small compared to its private counterpart.)

```
( Cert:
  ( Local-Name: john.q.bonch )
  ( Value:
    ( Principal:
      ( Public-Key:
        ( RSA-with-SHA1:
          ( N:
            #B8D1C7A2EBCE4EDDD346162129A40451999B28AC7747AC91139B6F12089D-
            #9A7E6E0471936DBCC1EF1684C5769C4441681FC8FE81A6F5003CD4A6B97A-
            #E394F489 )
          ( E: #11 ) ) )
      ( Server-At: spruce.lcs.mit.edu oak.lcs.mit.edu ) ) ) )
```

Figure 2-6: The `Cert:` used to performance-test the library.

Forgetting the real cryptography overhead, these slow times can be blamed on the lack of optimization in the library's design. A very fast design was not developed for two reasons: first and foremost, a heavily optimized library would be very difficult to understand, and, in any case, time constraints prevented any thoughts of fancy, fast code. Something that was easy to understand and debug, and would be finished soon, was the goal throughout library development.

2.7.1 Retrospection

As the performance problems and, to a certain extent, the form of the code itself show, this library was not written exactly as I would write it now. I had some very specific wishes for this implementation that crept up over the months it took to write it, that I couldn't follow. Given more time, I would have liked to add the following:

1. Group membership hints. This is the only significant part of the SDSI specification missing from this library (another part being the RFC822 address-to-**ref:** translation). The group membership check function only returns *true*, *false*, or *fail*, with no credentials hint accompanying the latter two indications. A truly complete library would include code to both assemble these hints, and to use them in preparing followup queries.

2. More intelligent object scanning. There are some bugs in the way the library currently scans objects from character streams, especially when it comes to strings, and fragments of strings. The scanning routines also read a character-at-a-time out of an input stream, which isn't efficient at all (especially when the input stream is really a network socket).

3. Faster object printing. Like scanning, the library isn't very quick about displaying objects. On the surface, this doesn't appear to be that important, but since printing objects happens quite a bit "under the hood" of the library, this does contribute to speed problems.

4. Better file format used in the cache and localities. The cache objects and the locality files are not stored in a particularly efficient manner - objects are "printed" to disk files in their SDSI canonical form. It takes quite a while to scan in elements of the cache and localities.

5. More memoization in the cache manager. Besides caching locality contents, it might be a win for the cache manager to remember even more specific sets of objects - those sets that satisfy certain queries.

6. Faster template matching. Perhaps a loss of generality in exchange for more speed would be a good idea. This could be achieved immediately by writing a better `sdsi_find_attribute_value`.

7. Smarter setting of current principal throughout an object. Currently, this is done every time an element is added to an object being built by the `sdsi_add_` functions. This doesn't need to be done for *every* element added, really - just after the *last* one, and the actual act of setting could probably be made faster.

8. Better error reporting. Currently, error messages are short, and while they do give a message describing a problem that caused the error, there is no larger context given to appreciate the message in. For example, if deep in some operation riddled with `ref:` evaluations, one of the evaluations fails, it would be

nice if the library included the offending `ref:` in an error message. Something like an *error context* structure in the library, where library functions put objects involved in errors, would be a good thing.

9. Faster error-handling primitives. By the primitives, I mean the `SETJMP` and `LONGJMP` macros used by the library's prologue and error macros. Currently, the library uses the system's standard `setjmp` and `longjmp` functions. A much faster library would feature specialized assembly-language functions for this task[5].

10. A consistent function naming scheme. About halfway through, I changed from a `sdsi_`*verb_action* form to a `sdsi_`*module_verb_action* form. It's not a big deal, but I think the latter style would be better for this library.

[5]For the curious, `lib/os_lib.h` contains prototype definitions for `gcc` on SPARCs, that I couldn't quite get working.

Chapter 3

The SDSI Server

The SDSI object server, (also called the SDSI server, or just *the server*) is a program that answers SDSI network queries. This program turns a workstation into an example of the most tangible and essential elements of the SDSI infrastructure. Principals rely on the servers they are registered with to provide their SDSI *presence*; they are listed in a `Principal:`'s `Server-At:` field, and

> ... 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.

The server is a short program, at just over two hundred and fifty lines. It is based on the library, and is a good example of a library client. Because it is so short and does so little outside of making library calls, it is extremely bound to this library's particular implementation. The library does so much of the work of actually *being* a server, that few design choices needed to be made for the server program, and those that were left were tied directly to the features and general disposition of the library. As a result, much of this chapter is not of real interest except to those making use of this particular implementation.

This server serves the three standard protocols from the SDSI specification: the `Get`, `Reconfirm`, and `Membership` protocols. These three protocols make basic certificate and membership information distribution possible. A fourth protocol, not in the original specification, was designed and added to fill an important void - the `Update` protocol exists for principals registered with a server to update their databases on that server. This protocol is described in Section 3.1.

Several notes on the actual operation of a SDSI server running this program are offered in Section 3.2. Many of them, like those that concern cache localities, are specific to the design choices made in the library. A small handful are generic, and mostly have to do with principal registration on servers.

3.1 The `Update` protocol

The SDSI specification says that "a server for a principal can be viewed primarily as an agent with a database of statements signed by that principal," but it fails to say how that database is managed by the principal. No protocol or procedures are given for a principal to use to add or remove certificates from that database. The `Update` protocol provides this functionality.

The `Update` protocol is completely implemented in the library, as are `Get` and the other SDSI protocols. It is not discussed in Chapter 2 with the rest of the library, because it seemed more fitting to discuss a protocol so specific to a server's operation in the chapter about the server.

The `Update` protocol exists to allow a user registered with a server to do two things:

- Add new certificates to his database on that server. This strictly *augments* that database - the protocol and the server make no attempt to remove or demote older certificates that are superseded in *any* sense whatsoever by new certificates.

- Remove certificates from the database on that server. This can only be used to remove *specific* certificates previously issued by the principal.

Of those two, removing certificates is the more interesting operation. To be specific, when a principal removes certificates from a server, they are not so much removed, as are their signatures *revoked.*

This seems rather contradictory to SDSI, since its online signature reconfirmations are supposed to eliminate the need for signature revocation lists. This is true at large, within the infrastructure, but an individual principal's servers, since they are that principal's proxy in the infrastructure, *do* need to have a list of signatures that have been revoked by the principal.

Without these local signature revocation lists, if a server merely "forgot about" and deleted revoked certificates from its cache to express their revocation, problems could happen. One example of such a problem is what would happen when one of a principal's revoked certificates came back to the server in some client's request, as credentials for that client. Assuming the signature (aside from really having been revoked) was still valid, the server would have nothing to "remind" it that it has been revoked - it could just be a certificate that its client issued that it has never seen before[1], *or* it could be a revoked certificate. Signature revocation lists are necessary on the server as state, to prevent clients' revoked certificates from being wrongly accepted.

The `Update` protocol is a simple two message command-response protocol. The command message is called an `Update.Order`, and it is a signed request that contains an `Update` section, a `Revoked:` section, or both. The `Update` section contains any new certificates that should be added to a user's database. Any `Revoked:` section contains the previously issued `Signed:` objects that should now be considered revoked.

Figures 3-1 and 3-2 contain two example `Update.Orders`. While any inner `Cert:` and `Signed:` objects within the orders represent valid signatures, these entire protocol messages will still be signed, for integrity, by the library when they are actually submitted to servers - and they will be signed with *very* short lifetimes, to prevent against replay attacks.

[1]This can especially happen when the certificate is a `Membership.Cert:` that the client principal issued and never updated his servers with, but then wanted revoked.

```
( Update.Order:
  ( Update
    ( Cert:
      ( ACL: ( read: ALL! ) )
      ( Local-Name: enemy )
      ( Value: ( Principal: ( Public-Key:
                              ( RSA-with-MD5:
                                ( N: #D2AA328CB2BF02 ... )
                                ( E: #11 ) ) ) ) ) )
      ( Signed:
        ( Date: 1997-03-18T15:09:37.625-0500 )
        ( Re-confirm: ( Period: PT5M ) )
        ( Signer: ( Principal: ( Public-Key:
                                 ( RSA-with-SHA1:
                                   ( N: #B8D1C7A2EBCE4EDDD3 ... )
                                   ( E: #11 ) ) )
                    ( Server-At: oak.lcs.mit.edu spruce.lcs.mit.edu ) ) )
        ( Object-Hash: ( SHA1 #2F623FEE40 ... ) )
        ( Signature: #B059111A80A5ACE5C35B ... ) ) ) ) )
```

Figure 3-1: An Update.Order that adds a new certificate.

```
( Update.Order:
  ( Revoked:
    ( Signed-Object:
      ( Signed:
        ( Date: 1997-03-18T14:55:54.734-0500 )
        ( Re-confirm: ( Period: PT5M ) )
        ( Signer: ( Principal: ( Public-Key:
                                 ( RSA-with-SHA1:
                                   ( N: #B8D1C7A ... )
                                   ( E: #11 ) ) )
                    ( Server-At: oak.lcs.mit.edu spruce.lcs.mit.edu ) ) )
        ( Object-Hash: ( SHA1 #95E76B ... ) )
        ( Signature: #9B9F783739DF7A37 ... ) ) ) ) )
```

Figure 3-2: An Update.Order that revokes a signature.

The order in Figure 3-1 submits a new, signed, name-binding `Cert:` to the server. Figure 3-2 is an `Update.Order` that revokes a previously-issued `Signed:` object from the server, which will effectively remove the object it signed - usually some kind of certificate - from the client's database on the server.

The `Update.Reply` message is extremely uninteresting. It is nothing more than a list message with the word `Update.Reply` signed by the server that received the order.

The code that runs on servers that handles `Update.Orders` takes care to only service orders from principals that are registered with the server. How principals become registered with servers is discussed in the next section.

3.2 Operation notes

There are not too many details surrounding running a SDSI server based on this implementation. Three things concern an operator of one of these servers: the creation of the server, the registration of new users on it, and regular maintenance of its object cache. Most of these operation notes are specific to this implementation.

To create a new server, first a principal for the server must be generated. This is done using the `pgp-to-sdsi` tool discussed in Chapter 4 to convert a PGP private keyring created just for the new server into a SDSI principal definition. This principal definition should then be placed in a new directory, created to contain it and the directories for the server's object cache and localities.

Registering new principals as clients of the server involves using the SDSI shell or its user interface [1], to sign and cache new membership certificates as the server principal. Each new `Membership.Cert:` will assert that one new client principal is a member of a special group belonging to the server[2]. This special group exists so the library has a way of knowing the set of principals allowed to submit `Update.Orders` to the server, and hence knows who is a proper registered client of the server.

[2]The actual group name is `sdsi-server-clients`. For the curious, this is maintained as a preprocessor symbol in the file `sdsi.h` of this implementation.

Once a server's principal has been created and some clients registered, the server can be brought up on the Internet. The server program, called `sdsid`, is a single, standalone binary (see Section 3.3 for more information on why) that is launched at the command line. It listens for `Get`, `Reconfirm`, `Membership`, and `Update` queries on TCP port 2500 on all network interfaces.

The only remaining operational decision is: how often should the server's localities be rebuilt? For good performance, the SDSI library makes use of the externally-built cache associations called *localities*, introduced in Section 2.3.1. This particular server uses the *coarse* cache searching behavior defined in that section, meaning that to satisfy queries, it searches localities exclusively, and ignores actual cache contents.

So, as new objects are added to the server's cache through the `Update` protocol, they will not be reflected in responses from the server until the localities are next rebuilt. Localities are rebuilt using the `sdsi-cleanup-rebuild` program, discussed in Chapter 4, but as the rebuilding process is disk- and CPU-expensive, the frequency of the rebuilds has to reflect a balance between this fact and the desire for a server that is appropriately up-to-date.

No suggestions are available to help decide how often cache localities should be rebuilt. The decision depends on the combination of a server's cache size, CPU power[3], and desired "reflection time" for new submitted certificates. These are all factors that depend on situation, and no trials were run for any significant combinations to offer any hints here.

Once a suitable locality-rebuilding frequency has been chosen, all that has to be done is invoke `sdsi-cleanup-rebuild` at the appropriate times.

3.3 Performance notes

This SDSI server, `sdsid`, is designed to run *monolithically* on a workstation, meaning that it is always running, as one single-threaded process that never forks, answering

[3]Although it is interesting to note that if a SDSI server's localities are stored in a network filesystem, they can be rebuilt by a machine other than the server itself.

all queries addressed to the server. The monolithic decision was made for two reasons:

1. While the localities, even when on disk, make for faster cache searches, for maximum performance they are kept in memory whenever possible, since scanning them in from disk incurs some cost (Section 2.3.1). One long-lived server process only has to scan a locality in when it is first needed, or when it changes on disk; if a new server process were started to answer each individual query, relevant localities would have to be scanned in, guaranteed, for every query.

2. The image of the server process in memory is typically large, and contains sensitive information, like the server's private key. The former is true because it does store localities in memory, as mentioned above, for performance. Forking a process of this size may incur a significant penalty on some systems, and at least always increases the amount of virtual memory that addresses sensitive data. Not forking prevents any such penalty, and also keeps the number of processes that carry sensitive information to the absolute minimum.

Query service time on a SDSI server with full locality support, then, is not a function of the number of principals the server serves for. Rather, a query's service time is related to the sizes of the localities associated with the principals on the server that are involved, directly or indirectly, in the query.

For example, a simple query addressed to a client of a server, that involves no credentials from other clients of the same server, will only involve the speaker locality associated with the client.

If the query relies on credentials from another client on the same server, however, the speaker locality of that client is also involved in answering the query (even if sufficient credentials were included directly in the query). This last fact is a result of the way that cache searches are run by the library - the credentials in the query *will* be found first, but if the library has another place it can look in for more, possibly relevant signed statements from the same individual, (in this case, an available speaker locality, since the credentials came from a local client) it will look there.

```
( Cert:
  ( ACL: ( read: ALL! ) )
  ( Local-Name: dude39 )
  ( Value: buddy39 ) )
```

Figure 3-3: The thirty-ninth certificate of a principal with 100.

It is really quite hard to give real numbers about query service times, since those times do depend so heavily on the size and number of certificates issued by the principal being queried. The only numbers available from testing involve querying a principal with 100 certificates, all similar to the thirty-ninth such certificate in Figure 3-3.

Two tests were run, both making a query for this certificate. One test made the query to a server that had just been started, and so had no localities loaded into memory, and the second made the query to a server that was known to have the queried principal's locality in memory - merely the same server that had just passed the first test. The server in question was an unloaded Sun SPARCstation 5 with 28MB of memory.

The performance numbers for the server were, like the performance numbers for the library (Section 2.7), uninspiring. The first query took almost 75 seconds, and the second query took 34 seconds. These times include everything the server did to service the query, from verifying the signature on the request, to performing the certificate search, to signing and returning the result.

The difference of forty seconds can be completely attributed to the amount of time the server, during the first test, took to read the queried principal's locality into memory. The size of this locality was 70K on disk. The "real" time of 34 seconds includes searching through the loaded locality with the template, searching through it again to pick up the `Delegation-Cert:` needed to sign the reply, and finally signing and returning the reply.

Not much more can be said about the server's poor performance, except that many approaches taken in designing the library were less than optimal, from a speed perspective. Section 2.7.1 discusses them. If the library were appropriately redesigned, the server's performance could be much better.

Chapter 4

Supporting tools

The SDSI library and server from this implementation do not make a complete and usable SDSI package. Many necessary support functions, some implied by features present and missing from the library, are still needed. These include:

- Key generation. A SDSI server or user has to create a principal (i.e., a public/private key pair) in order to do anything. The library and server provide no key generation routines.
- Cache and locality maintenance. The SDSI library only knows how to grow its object cache, not shrink it, and how to use localities, not make them. Left to its own devices, the library will grow its object cache forever and never update the associated localities.
- A real user interface. With servers running, but nothing in the hands of the users except the library, SDSI would not be usable. Individual library programs would have to be written and compiled for every task.

These missing items are provided as additional support programs, all based on the library. For key generation, one tool leverages off of PGP's key generation ability, and merely converts PGP key rings into certificates and principal definitions. `pgp-to-sdsi` is discussed in Section 4.1.

`sdsi-rebuild-cleanup`, covered in Section 4.2, is a program that cleans "dead" objects out of the cache, and rebuilds the cache localities.

Most important of all, a program that is the foundation for a user interface, `sdsish`, is introduced in Section 4.3. This "SDSI shell" is a command-line interface to the SDSI library, and allows the user to type commands to manipulate, sign, and encrypt SDSI objects and execute queries. It also has hooks in it to enable a larger, more user-friendly World Wide Web interface[1] to execute its shell commands automatically.

4.1 The PGP-to-SDSI converter

The PGP-to-SDSI converter is quite possibly the most significant of the tools, because, as its name implies, it forges a link from PGP [2] to SDSI. It can do two things: given a PGP private key ring, it will make a SDSI principal definition (Section 2.1.6), and given a PGP public key ring, it will make crude SDSI name-binding certificates, that the user can finish and sign at his convenience.

The former ability eliminates the need for the library to have its own public/private keypair generator, since it effectively uses the well-written one already included in the PGP distribution. Plus, if someone is already using PGP, they can continue to use the same public key in SDSI.

The latter ability, to convert a PGP public key ring into preliminary name-binding certificates, is even more important. This allows a user to automatically convert the entries of people he already knows closely in his PGP "web of trust" into entries in his initial SDSI local namespace. When those other PGP-savvy friends, relatives, and colleagues do the same, immediately a sizeable number of SDSI linked local namespaces will exist.

Virtually no choices were involved in the design of this converter; once the PGP keyring format[3] was found, writing the converter was more of an exercise in using and an opportunity to test the cryptographic routines in the SDSI library. When converting members of a public key ring, nothing is actually encrypted - the elements of each member (the username, and RSA modulus and public exponent) are merely turned into octet-strings, and used to build up a `Cert:` that binds the username to

```
% pgp-to-sdsi ~/.pgp/pubring.pgp
PGP version number in key certificate is 3.
crude unsigned certificate for "alyssa.p.hacker":

( Cert:
  ( Local-Name: alyssa.p.hacker )
  ( Value:
    ( Public-Key:
      ( RSA-with-???:
        ( N: #CDF44F325A9DF4 ... )
        ( E: #11 ) ) ) ) )
done.
%
```

Figure 4-1: An example of `pgp-to-sdsi` converting a PGP public key ring.

a principal based on that public key.

For example, Figure 4-1 shows `pgp-to-sdsi` converting a public keyring with one element. It yields a name-binding certificate that the user can finish (by replacing the `???` with the name of the signing hash that the principal actually uses), and then sign and distribute to his servers.

When converting a user's private keyring, a sealed private RSA exponent is found alongside the user's public RSA modulus in the ring. The exponent is sealed in an IDEA key derived from the user's passphrase. After the user provides the passphrase, this exponent is unsealed and a SDSI `Private-Key:` object is built around it. The `Private-Key:` is then resealed in the original IDEA key and added to the user's new principal definition, which also holds a `Principal:` constructed from his public key.

Once generated through this process, a user's principal definition can be stored in a file and be used in interactions with the SDSI shell or user interface [1]. Normally, this is the case, and the user will proceed immediately to use the shell or user interface to sign and issue the selected certificates formed from his PGP public keyring.

Or, if the principal definition was newly created just for use as the principal that will "be" a SDSI server, the new server can now be brought up, and configured. More about operating SDSI servers is found in Chapter 3.

4.2 The cache cleaner and locality rebuilder

The tool that cleans SDSI object caches, and rebuilds localities in the process, is called `sdsi-rebuild-cleanup`. As its name and the previous sentence hint, this program does two things:

1. It removes elements from the permanent cache that are found to be "dead". A cache element, strictly speaking, is a top-level protocol message that was received, or a locally-generated certificate, that was written into its own file into the cache. A cache element is dead when it contains no unexpired signatures and none of its object or subobjects are signed by an unexpired signature.

2. It rebuilds the cache localities. Localities, discussed in Section 2.3.1, are specialized subsets of the objects in the cache, used to speed cache searches. When the cache grows with new objects, the library does not update any of the localities - for the new cache objects to be reflected in the localities, the latter have to be rebuilt.

This is by no means a glamorous tool; it's a C program based on the library that is not pretty to look at, and it has some shortcomings. It works by first finding the cache and localities directories that the library uses through environment variables (see Section 2.1.7 for a discussion of how common filesystem elements are specified to the library), and walks the elements in the cache, building a new localities directory near the old one.

The cache is walked in two passes. Pass one finds all `Signed:` objects anywhere in the cache that have not expired. Pass two loops over all element objects and subobjects, building the *speaker*, *object*, *key*, and `Signed:` localities it finds along the way, in this manner:

- A subobject gets added to the speaker locality associated with each principal who has signed it. A speaker's speaker locality filename is based on the MD5 hash of the speaking `Principal:`, and this file ends up containing every object in the cache that the principal has signed.

- If a subobject is a `Signed:`, it gets added to an object locality associated with the object it signs. The filenames of an object's object locality files (there can be several of them) are based directly on the `Object-Hash:`s included in the `Signed:`s that sign them. This means that an object's object locality is a collection of files by hash function, with each file containing all signatures on the object that used that hash function.

- If a subobject is some kind of `-Key:`, it gets added to the corresponding key locality, depending on whether it is a public, private, or shared-secret key.

- `Signed:` localities are not actually separate localities - they are more conceptual. As described in Section 2.3.1, a `Signed:`'s locality is a combination of the part of the object locality its `Object-Hash:` makes it a *member* of, plus the object locality that is *associated* with *it*. (The two are different - the latter is a locality that includes all of the `Signed:`s that sign the original `Signed:`).

While pass two walks the cache, building these localities, any cache elements that are not found to contain live signed data, are removed. When `sdsi-rebuild-cleanup` is done, it removes the old localities directory and replaces it with the new one it just finished.

4.2.1 Known bugs

There are some known bugs in `sdsi-rebuild-cleanup`.

1. It assumes that the user is using localities. Currently, it can't be used to merely clean the cache of a user that *doesn't* use localities. If a localities directory can't be found, it will dump core; the temporary solution is to create a localities directory, run `sdsi-rebuild-cleanup`, and then remove the resulting new localities directory.

2. When `sdsi-rebuild-cleanup` is finished rebuilding the new localities in its temporary directory, it swaps out the old localities directory and swaps in the temporary one under the correct name. To prevent insanity for the duration of

the two system calls necessary to do this, it is supposed to send a `SIGHUP` to a running `sdsid` (the SDSI object server), to make it pause for 4 seconds while the swap happens. `sdsid` writes its process ID to a file in `/tmp` for this purpose, but `sdsi-rebuild-cleanup` never checks for it, reads it, and sends the signal.

3. Pass one just looks for signatures that indicate that they haven't expired yet; it doesn't actually check to see that signatures are cryptographically *valid* - i.e., `sdsi-rebuild-cleanup` never confirms the contents of `Signed:`s' `Signature:` fields.

4.3 The SDSI shell

The SDSI shell, `sdsish`, is absolutely the most useful SDSI tool. `pgp-to-sdsi` is important; this shell is useful. It is a basically nothing except a command line interface to SDSI library functions, but the fact that such an interface exists means that people can *use* SDSI.

The shell allows a human user to create and manipulate objects, manage his object cache and server databases, and perform all sorts of queries. The command environment is rather flexible: named variables are used with all commands, and command results are returned in new named variables.

`sdsish` also serves to support the SDSI Web-based graphical user interface[1]. It does this by running as a network daemon, allowing certain locally-running Perl scripts to connect to it and issue shell commands. It also runs the Web server and Web browser that, with those scripts, complete the GUI architecture.

4.3.1 Command line mode

The command line mode is started by invoking the shell as `sdsish`. Using the principal definition file as found by the library (see Section 2.1.7 for a discussion of how the library finds common filesystem elements), it asks the user for the passphrase that unseals the definition's private key, then presents a simple prompt:

```
sdsish>
```

From this prompt, many different SDSI operations are possible. There are several commands currently available; Figure 4-2 shows the output of the `help` command listing them.

All commands take their arguments, with the exception of their integer arguments, through named variables - SDSI objects are never given literally on the command line. Variables are introduced with the `define` command, and can be displayed with the `print` command. Examples of how variables are created and displayed are in Figure 4-3.

Once variables have been defined, commands are given to work on them. There are enough commands to prevent discussion of each one here, but a help facility has been built into the shell: `help` *command-name* will bring up a few sentences on how to use the named command.

One example of issuing a command is given in Figure 4-4. It shows a variable being defined to hold a `Cert:` object, and the `sign` command being used to sign it[1]. Note how the result of the `sign` is both displayed and assigned to a new named variable, `$0`. A new "dollar temporary" is defined to hold the result of any command that returns some value, to allow the use of that value in subsequent commands.

4.3.2 Graphical user interface mode

The previous section describes the command line mode of `sdsish`. The shell also has a *graphical user interface* (GUI) mode, which exists to support Gillian Elcock's Web-based user interface[1], by providing its command set to Perl scripts triggered by a user's interactions with a Web browser.

The GUI mode is started by invoking the shell as `sdsiui`. When run under this name, the shell never puts up a command line for the user, but instead bootstraps the

[1]The `2` argument to `sign` asks for an appendix mode signature on the object being signed. This value corresponds directly to the `flags` value passed in to the `sdsi_sign_object` library function; refer to the SDSI library documentation[8] for more information on this and other functions' possible flag values.

```
sdsish> help

command summary:

help or ?          - displays help
define             - defines a new variable
print              - prints a variable
get                - executes a Get query
lookup             - performs a namespace lookup
hash               - hashes an object
eval               - evaluates an object
eval-to-cert       - almost-evaluates an object
set_cp             - sets an object's current principal
force_cp           - forces an object's current principal
sign               - signs an object
confirm            - confirms an object's signers
verify             - verifies a Signed: object
encrypt            - encrypts an object
decrypt            - decrypts an object
cache              - adds an object to the cache
uncache            - deletes a cache subset
car                - returns the car of a list object
cdr                - returns the cdr of a list object
cons               - returns the cons of two objects
update             - updates your servers
width              - sets the display width
dollar             - sets the next $ temporary number
member?            - executes a Membership query
equal?             - compares two objects

sdsish>
```

Figure 4-2: The list of `sdsish` commands.

```
sdsish> define my-cert
remember a lone string definition must end with a dot on a line by itself.
( Cert: ( Local-Name: favorite-color ) ( Value: blue ) )
sdsish> print cert
( Cert: ( Local-Name: favorite-color ) ( Value: blue ) )
sdsish> define car-make
remember a lone string definition must end with a dot on a line by itself.
Volkswagen
.
sdsish> print car-make
Volkswagen
sdsish> define my-cert
remember a lone string definition must end with a dot on a line by itself.
( Cert: ( Local-Name: car-make ) ( Value: Volkswagen ) )
sdsish> print my-cert
( Cert: ( Local-Name: car-make ) ( Value: Volkswagen ) )
sdsish>
```

Figure 4-3: Examples of creating and displaying variables in `sdsish`.

GUI and makes that command line available to the GUI's Perl scripts. This involves the following:

1. A child process is started that will listen for HTTP connections from a Web browser. The TCP port it listens on is assigned at random by the operating system, out of the set of available ports. Each incoming connection is passed to a freshly-invoked HTTP server.

   Currently, the only supported HTTP server is NCSA's `httpd`. The specific one to use is found in the `gui` directory of the user's SDSI directory (see Section 2.1.7 for a discussion of how the library finds common filesystem elements) or indicated explicitly by the environment variable `SDSI_GUI_HTTPD`. The configuration file for `httpd`, `httpd.conf`, is found alongside it, or is indicated by the environment variable `SDSI_GUI_HTTPD_CONF`.

2. A cryptographic nonce, called a *cookie*, is generated. This nonce will be used by the GUI Perl scripts to authenticate themselves to the shell when they connect to it, so that they can issue shell commands to sign certificates, etc., as the

```
sdsish> define cert
remember a lone string definition must end with a dot on a line by itself.
( Cert: ( Local-Name: favorite-color ) ( Value: blue ) )
sdsish> sign cert 2
( Cert:
  ( Local-Name: favorite-color )
  ( Value: blue )
  ( Signed:
    ( Date: 1997-05-13T13:12:20.738-0400 )
    ( Signer:
      ( Principal:
        ( Public-Key:
          ( RSA-with-SHA1:
            ( N: #CDF44F325A9DF471 ... )
            ( E: #11 ) ) )
        ( Server-At: spruce.lcs.mit.edu ) ) )
    ( Object-Hash: ( SHA1 #E7AB87437B ... ) )
    ( Signature: #844BCF484339DFD9 ... ) ) )
that returned value has been named $0.

sdsish>
```

Figure 4-4: An example of using the `sign` command in `sdsish`.

current user. Two parts of this cookie are actually the TCP port on which the shell will be listening for connections from the Perl scripts, and the TCP port that the HTTP server child is listening on.

3. A Web browser is started to display the GUI's home page, or a currently running browser is signaled to load it in. Like the HTTP server support, only one Web browser, Netscape, is currently recognized by the shell.

   The page that the new or existing Netscape is pointed to is actually a small HTML "cookie" page. Stored in `/tmp`, and readable only by the current user, this page instructs the browser to load the real GUI home page through an initial CGI script, passing it the cookie. Only scripts invoked from pages tracing their "heritage" back to this original CGI script invocation will have the cookie necessary to connect to the shell, authenticate to it and issue commands.

4. The shell then waits for connections from the GUI Perl scripts. A connection is closed immediately unless the remote peer is on the local machine and can present the correct cookie. This prevents connections from other parties from being able to present commands to the shell and have them executed as the current user.

Once a connection from a GUI Perl script is successfully established, the shell executes the commands presented to it, and returns their results. More detail about the GUI, its architecture, and the authentication mechanism described loosely above, can be found in [1].

4.3.3 The code, adding new commands

The code that makes up **`sdsish`**, like that for **`sdsi-rebuild-cleanup`**, isn't the prettiest. This is because `sdsish` was originally conceived more as a tool for testing, and not as a prime-time program. Its command parsing, type checking, and symbol table management are all very quick-and-dirty. For these reasons, its actual design is not documented very much here.

```
{ "get", do_get, "ooI",
  "executes a Get query",
  "usage: get <principal> <template> [<flags>] -  queries <principal>\n" \
  "  for objects it has signed that match <template>.  see sdsi.h\n" \
  "  for a listing of the possible bits in <flags>.\n" }
```

Figure 4-5: The `sdsish` command declaration for the `get` command.

However, one goal of `sdsish` was to make it very easy to add new commands to it. The file `sdsish_commands.c` contains all of the functions that implement the various shell commands. Every function that implements a command has the same prototype: `something *` *function-name*`(something *);`

A `something` is the data structure used to pass values of many types within the shell. A `something` can represent a SDSI object, SDSI list, SDSI octet-string, a simple C string, or an integer. `somethings` can made into a linked list; this is how multiple arguments are passed into a command function. Command functions optionally return a single `something` containing the result of the command. A `something` indicates what type of thing it *actually* represents, then includes either a pointer to or the actual value of that thing.

Adding a new command to the shell involves the following tasks:

1. Decide on the name for the new command, and the name of the C function that will implement it. Add a new prototype declaration using the `FUNC` macro to the top of `sdsish_commands.c` using the name of the new function.

2. Add a new command declaration to the `commands` array. `commands` is an initialized array of type `command` - a data structure that holds a command declaration.

   The `command` structure used to declare the `get` command is given in Figure 4-5 as an example. Its fields are:

   (a) The name of the command.

   (b) The name of the function implementing the command.

Character	Represents	Character	Represents
o	SDSI object	l	SDSI list
s	SDSI octet-string	i	integer
$	simple C string		

Table 4.1: `sdsish` command definition argument types.

(c) The command's argument list description. Its contents are described below.

(d) A short explanation of the command.

(e) Full help for the command.

A command's argument list description is a string, where each character represents an argument indicating the type of `something` that argument must be. Valid types are given Table 4.1. An `o`, `l`, `s`, and `i` may be capitalized: this causes a NULL object (or, in the case of an integer, the value zero) to be passed in as the corresponding argument if none is specified on the command line - in other words, a capitalized argument type specifies that the argument is optional.

3. Write the C function implementing the command. These functions are typically short, since all most of them do is extract pointers to the different C structures in the linked list of argument `somethings` passed in, hand them to a library function, and package the return value back into a `something`. The function implementing the shell's `get` command is a good example of this, and is shown in Figure 4-6.

```
/* the "get" command: */
something * do_get(args)
     something *args;
{
  sdsiObject *P, *template;
  int flags;
  something *ret;

  /* extract our arguments: */
  P = args->the_object;
  template =args->next->the_object;
  flags = args->next->next->the_int;

  /* make the call: */
  ret = new_thing(NULL);
  ret->the_list = sdsi_execute_get_query(P, template, flags);
  ret->what = A_LIST;
  return(ret);
}
```

Figure 4-6: The `sdsish` command function `do_get` for the `get` command.

Appendix A

Building and Porting SDSI

This SDSI implementation was developed in C on Sun SPARCstations running SunOS 4.1.4, and compiled with GNU's `gcc` version 2.7.2. Since portability was another goal while writing the library, server, and tools, this implementation should be fairly easy to port to other architectures, operating systems, and compilers. The code should be amenable to all Unix-like platforms, and (with the probable exception of the networking code) should even be portable to a DOS/Windows environment.

In this appendix are a description of the implementation's source hierarchy, instructions on how to build it on its native SPARC/SunOS platform, and notes on those parts that need to be modified in a port to another platform.

This appendix assumes the reader has already unpacked the SDSI distribution into a local directory; throughout, whenever */base* is used as part of an example pathname, substitute for it the fully-qualified path to that directory.

A.1 The SDSI source tree

Starting from the root of the SDSI source tree at `/base`, this implementation is split across five directories:

`include` This directory contains C header files that define the external (public) interface to the SDSI library. There is no `Makefile` in this directory; however, Section A.2 discusses the files that are constructed from library sources, and Section A.3.1 describes how to port the platform-specific files.

`lib` This directory contains the C header files that define the internal (private) interface to the SDSI library, the C code files that make up the library itself, and other table files that are processed when building the library.

`lib/crypto` This directory contains the set of directories that hold the "third-party" parts of the SDSI library: the cryptography modules. Since cryptography functions tend to be unusually machine-specific, where the chosen modules came from and their porting issues are covered in Section A.3.2.

`sdsid` This directory contains the C code file that is the SDSI object server.

`tools` This directory contains the C code files that are the SDSI support tools.

A.2 How to build the distribution

This is a step-by-step procedure for building the SDSI distribution just unpacked from a `tar` file. It is assumed that the current platform is a SunOS 4.1 SPARC with `gcc`, since the `tar` file contains the distribution configured to compile for that target.

1. Build the third party cryptography routines:

   ```
   % cd /base/lib/crypto
   % make all
   ```

   This builds all of the objects in the `md5`, `sha`, `des`, `idea`, and `gmp` directories under `/base/lib/crypto`. These objects will be added to `libsdsi.a` at the end of the next step:

2. Build the library:

   ```
   % cd /base/lib
   % make all
   ```

This creates the SDSI library. It begins by building four files from two tables in `/base/lib`. The tables are `sdsi_words.table` and `sdsi_errors.table`. These tables contain C preprocessor symbol names and strings that they are meant to correspond to, for the SDSI reserved words (Section 2.1.4) and error codes (Section 2.1.5. The helper script `make-table` transforms these into `/base/include/sdsi_words.h` and `/base/lib/sdsi_words.c`, and `/base/include/sdsi_errors.h` and `/base/lib/sdsi_errors.c`, respectively. The `.h` files contain numeric definitions for the preprocessor symbols, and the `.c` files have a corresponding string array definition.

Then, it builds all of the objects in `/base/lib` from their C sources, and combines those with the objects under `/base/lib/crypto` to make `libsdsi.a`.

3. Build the server:

   ```
   % cd /base/sdsid
   % make all
   ```

   This builds the SDSI server `sdsid` from `sdsid.c` and `libsdsi.a`.

4. Build the support tools:

   ```
   % cd /base/tools
   % make all
   ```

   This builds the SDSI tools `pgp-to-sdsi`, `sdsi-cleanup-rebuild`, and `sdsish` from C sources and `libsdsi.a`.

A.3 How to port the distribution

The SDSI library, server, and tools were all written with portability in mind. While it is generally impossible to create a large software distribution that will compile without modification even within the Unix world, the SDSI library should be reasonably portable.

All of the machine, operating system, and compiler dependencies that the SDSI library itself has have been isolated in three different include files: `/base/include/sdsi_arch.h`, `/base/include/sdsi_os.h`, and `/base/lib/os_lib.h`, each discussed individually here.

Instructions on reconfiguring the third party cryptography routines, and an explanation of how many of them are "glued" to and automatically use the library's machine dependencies, are in the last section.

A.3.1 The platform-dependent header files

`/base/include/sdsi_arch.h` This is a C header file containing information about the underlying machine's byte order and the specific C scalar types that it provides, matching the SDSI library to the specific platform's hardware and compiler. It contains definitions for C preprocessor symbols that describe the machine's byte order, primitive integer types, and non-specific pointer and handle types.

- Machine byte order is specified in the symbol `_SDSI_TYPE_32_BYTEORDER`. Its value is a 32-bit value, which, when stored as a 32-bit value on the target machine, will store from lowest-address to highest-address the bytes 4, 3, 2, and 1, respectively. For example, `0x4321` would specify normal big-endian, and `0x1234` would specify normal little-endian.
- Many symbols' names take the form `_SDSI_TYPE_`*X*`_`*Y*`INT`, and these are used as names for types of specific size and signed-ness. *Y* is always either `S` or `U`, to indicate a signed or unsigned type, respectively. When *X* is a numeric value, it specifies the total size, in bits, of the type. The only possible non-numeric values of *X* are `NAT` and `FAT`, which are used to specify the "natural" value size of the architecture, and the largest ("fattest") possible value size the architecture and compiler support.
- The non-specific pointer and handle types are defined in the symbols `_SDSI_TYPE_VOID_PTR` and `_SDSI_TYPE_VOID_PTR_PTR`. These names betray

the fact that, on all modern systems, the `void *` and `void **` types are available for these purposes. On systems that do not allow pointers to `void`, these symbol values should be `char *` and `char **`, respectively.

Currently, `/base/include/sdsi_arch.h` is populated with SPARC definitions, with some `__GNUC__`-sensitive sections that introduce 64-bit integers (which are available when using `gcc`, even on 32-bit native systems.)

`/base/include/sdsi_os.h` This is the second header file that continues to match the SDSI library to a specific platform, but now to features of its operating system and compiler. This file contains more C preprocessor symbols:

- The symbol `INTERFACE` is available to contain keywords that should be present in declarations or prototypes of SDSI library interface functions. This is to make those definitions conform to the calling convention that will be used with the library. Under Unix systems, this symbol will be defined with no value, since the calling convention is always fixed. However, when porting the library to Windows, `INTERFACE` might be defined to be `__far __export __pascal`, so the SDSI library can correctly become a DLL.
- The symbol `NOARGS` controls what appears in the argument section of prototypes and definitions of functions that take no arguments. On most modern systems, this is `void`. On some older systems, this may have to be empty.
- The symbol `PROTOTYPE` is a macro that controls how function prototypes are built for the compiler. The macro takes one argument - the argument list for the prototype. On compilers that support full prototypes (most modern ones do) this macro simply expands into that argument. When full prototypes are not supported, define this macro so that it expands into nothing, and simple prototypes will always be emitted to the compiler.

Currently, `sdsi_os.h` is populated with definitions for Unix systems running `gcc`.

`/base/lib/os_lib.h` This is another header file that contains platform-specific definitions for the SDSI library, except, unlike `/base/lib/sdsi_arch.h` and `/base/lib/sdsi_os.h`, it is private to the library - its definitions don't need to ever be included as any part of the external library interface.

The C preprocessor definitions in this file currently only provide symbols that name members of the operating system's equivalent of the Unix `setjmp` family. `setjmp` functionality, which should be provided in all modern C runtime environments, is needed for the catch-and-throw error handling facilities, which are described in Section 2.2.

- The symbol `JMP_BUF` represents the name of the data structure that holds a continuation. For Unix, this is just `jmp_buf` as defined in `<setjmp.h>`.
- The symbols `SETJMP` and `LONGJMP` map similarly to the names of functions that store the current continuation and jump to a continuation. For Unix, these map straight to `setjmp` and `longjmp`, respectively.

A.3.2 The cryptography routines

The compilation of cryptography routines is notoriously machine-dependent, because these routines always have compute in a precise fashion, regardless of the platform's byte order or native integer size.

There are five "third-party" cryptography modules contained in directories under `/base/lib/crypto`. Each module falls into one of two categories: either it contains a script or `Makefile` that takes care of configuring or compiling the module on the current platform, or, in the case of the smaller modules, its header file has been modified into a "glue" header file that gets information from the platform-dependent library files of the previous section. These latter modules don't need any further configuration in order to compile.

The five modules are:

`des` This is the popular `libdes` package written by Eric Young (`eay@mincom.oz.au`). It is not glued to the SDSI library, but should compile on most Unix machines with gcc without any work. If compiling on a Unix without gcc, edit /base/lib/crypto/Makefile to change the des: target to read make cc instead of make gcc.

des/INSTALL does contain instructions on work that needs to be done to compile this library on systems missing some runtime-environment support, or on 64-bit systems such as the DEC Alpha. It also lists desirable make targets for different Intel $x86$ platforms (making the library use its Intel assembly language routines) - a chosen target name should replace the gcc in the des: target body in /base/lib/crypto/Makefile.

gmp This directory holds the GNU multi-precision library, which is used to perform RSA's modular exponentiation. It is not glued to the SDSI library, but makes use of the popular GNU-ism configure.

The SDSI distribution comes with gmp configured for sparc-sun-sunos4.1.4. To prepare gmp for compilation on another platform:

```
% cd /base/lib/crypto/gmp
% ./configure
```

idea This directory contains the IDEA encryption code, lifted straight from Phil Zimmerman's pgp distribution. This module has been glued to the SDSI machine-dependent header files, so no configuration of this module is necessary. idea/idea.h, the glue header file, is a modified version of the original idea.h.

md5 This directory contains RSA's reference implementation of MD5. Like the IDEA module, this has been glued to the SDSI machine-dependent header files, so no configuration of this module is necessary. md5/md5.h, the glue header file, is a modified version of the original md5.h.

sha This directory contains a implementation of SHA derivative of one found in [11]. It has has been glued to the SDSI machine-dependent header files, so no configuration of this module is necessary. sha/sha.h, the glue header file, is a modified version of the original sha.h.

The glue header files are all modified versions of the original modules' header files, with the normally hand-configurable machine-specific section replaced by sections based on /base/include/sdsi_arch.h and /base/include/sdsi_os.h.

There are usually two such sections, chosen based on the presence of the C preprocessor symbol _SDSI_CRYPTO_PRIMITIVES. When these header files are being included by /base/lib/crypto-primitives.c, this symbol is defined, to select a set definitions sufficient to mate the SDSI library to the module. When this symbol is not defined, the header file is being included by the module itself for the purposes of its own compilation, so definitions needed to mate the module to the SDSI library are chosen instead.

Figure A-1 contains an example of the glue in sha/sha.h.

```
/* have to pull these in from explicit locations: */
#include "../../../include/sdsi_os.h"
#include "../../../include/sdsi_arch.h"

#ifndef _SDSI_CRYPTO_PRIMITIVES

/* this glues SHA to us: */
typedef _SDSI_TYPE_8_UINT BYTE;
typedef _SDSI_TYPE_32_UINT LONG;

#if (_SDSI_TYPE_32_BYTEORDER != 0x4321)
#define LITTLE_ENDIAN
#endif

#else  /* _SDSI_CRYPTO_PRIMITIVES */

/* this glues us to SHA: */
#define BYTE _SDSI_TYPE_8_UINT
#define LONG _SDSI_TYPE_32_UINT

#endif  /* _SDSI_CRYPTO_PRIMITIVES */

#define SHA_BLOCKSIZE 64
#define SHA_DIGESTSIZE 20

typedef struct {
    LONG digest[5]; /* message digest */
    LONG count_lo, count_hi; /* 64-bit bit count */
    LONG data[16]; /* SHA data buffer */
} SHA_INFO;

void sha_init(SHA_INFO *);
void sha_update(SHA_INFO *, BYTE *, int);
void sha_final(SHA_INFO *);

void sha_stream(SHA_INFO *, FILE *);
void sha_print(SHA_INFO *);

/* lose any glue we need to: */
#ifdef _SDSI_CRYPTO_PRIMITIVES
#undef BYTE
#undef LONG
#endif  /* _SDSI_CRYPTO_PRIMITIVES */
```

Figure A-1: The glue in the header file sha/sha.h.

Bibliography

[1] Gillian D. Elcock. A web-based user interface for a simple distributed security infrastructure (SDSI). Master's thesis, M.I.T., May 1997.

[2] Simson Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly & Associates, 1994.

[3] Arne Helme. *File formats used by PGP 2.6.* `http://www.gildea.com/pgp/pgformat/pgformat.html`, May 1994.

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

[5] Xuejia Lai. On the design and security of block ciphers. In J.L. Massey, editor, *ETH Series on Information Processing*, volume 1. Hartung-Gorre Verlag, Konstanz, Switzerland, 1992.

[6] David Gadbois, et al. *GC FAQ – draft.* `http://www.centerline.com/people/chase/GC/GC-faq.html`.

[7] Matthew Fredette. *A SDSI 1.0 Implementation.* `http://theory.lcs.mit.edu/~cis/sdsi.html`, (to appear) 1997.

[8] Matthew Fredette. *SDSI Library Documentation.* `http://theory.lcs.mit.edu/~cis/sdsi/library-documentation.html`, May 1997.

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

[10] Ronald L. Rivest and Butler Lampson. SDSI - a simple distributed security infrastructure. `http://theory.lcs.mit.edu/~rivest/sdsi10.html`, September 1996.

[11] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, 1996.

5660- 51