SNP: An Interface for Secure Network Programming*

Thomas Y.C. Woo, Raghuram Bindignavle, Shaowen Su and Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

SNP provides a high-level abstraction for secure end-to-end network communications. It supports both stream and datagram semantics with security guarantees (e.g., data origin authenticity, data integrity and data confidentiality). It is designed to resemble the Berkeley sockets interface so that security can be easily retrofitted into existing socket programs with only minor modifications. SNP is built on top of GSS-API, thus making it relatively portable across different authentication mechanisms conforming to GSS-API. SNP hides the details of GSS-API (e.g., credentials and contexts management), the communication sublayer as well as the cryptographic sublayer from the application programmers. It also encapsulates security sensitive information, thus preventing accidental or intentional disclosure by an application program.

1 Introduction

The explosive growth of network connectivity has significantly aggravated the problem of security. Most existing network programming paradigms adopt a trust-based approach to security (e.g., trusting network packets, trusting hosts). This is no longer adequate, especially for malicious attacks. Indeed, with easy access to networks and availability of sophisticated network tools, the effort to mount attacks such as spoofing network packets or sniffing illicit information from network traffic is substantially reduced. To effectively counter these attacks, a coherent security infrastructure is needed. An important element of such a infrastructure is a convenient abstraction for secure application network programming.

In recent years, distributed systems security has received a great deal of attention. For example, a number of authentication systems (e.g., Kerberos from MIT [15], SPX from DEC [17] and KryptoKnight from IBM [9]) have been designed and implemented. Although these systems do provide an adequate solution for typical network security concerns, they suffer a major common drawback, namely, it is difficult to integrate them into an application. More specifically, they do not export a clean and easy-to-use interface that can be readily used in implementing a distributed service. For example, it often takes a considerable amount of effort to "kerberize" an existing distributed service. Besides, the interface provided is not portable, making the switch from one authentication system to another a non-trivial task.

The recently published Internet draft standard *Generic Security Service Application Program Interface* (GSS-API) [8] alleviates the problem somewhat. In fact, both SPX and KryptoKnight[1] have already implemented a small subset of GSS-API.[2] However, the GSS-API interface is still too low-level to be practical for typical network application programming. Its proper use requires intimate understanding of the underlying GSS-API concepts, which can cause significant distraction from the main task of a program. It is valid to say that GSS-API is more suited for use in system software than in regular application programming. Indeed, it is intended that a typical caller of GSS-API be a communication protocol, e.g., telnet, ftp [8, p. 2].

We believe that what is needed is an abstraction for secure network programming that can hide most of the details of GSS-API while retaining the same ease of use as most existing abstractions for network programming. As an analogy, the raw interface to a protocol (e.g., `tcp_input()/tcp_output()` for TCP) is often difficult to use, whereas programming using a higher-level abstraction (e.g., sockets, TLI) is significantly easier.[3]

In this paper, we discuss the design and implementation of SNP (*Secure Network Programming*), a high-level

*Research supported in part by NSA INFOSEC University Research Program under contract no. MDA 904-91-C7046 and MDA 904-93-C4089, and in part by National Science Foundation grant no. NCR-9004464.

[1] KryptoKnight is not public-domain. The use of GSS-API is mentioned in [9]. But it is not clear to what extent the interface has been implemented.

[2] A recent article in *comp.protocols.kerberos* states that implementations of GSS-API for Kerberos will also be available.

[3] In fact, Berkeley sockets have often been touted as a major contributing factor to the popularity of TCP/IP. Although Berkeley sockets can support a variety of protocols, it was designed mainly with TCP/IP in mind.

abstraction for secure network programming that we have developed. SNP is like sockets or TLI in that it is an interface that provides applications access to network communications. However, it differs from sockets or TLI in many significant ways:

- SNP provides *secure* network communication. For example, it provides data origin authenticity, data integrity and data confidentiality services on top of the usual *stream* and *datagram* services provided by sockets or TLI. The precise services provided by SNP are detailed in Section 4.
- SNP provides an end-to-end communication abstraction at the application level, whereas sockets and TLI are transport level abstractions.[4] More specifically, a socket represents a transport level endpoint (e.g., a TCP port), while an SNP endpoint represents an application layer entity (e.g., a server). This distinction is important and is further explained in Section 3.

SNP is implemented on top of GSS-API. It is currently in the form of a library. It adopts the same basic design as sockets (though several new calls have been added), which allows easy transitions from socket-based programs.

The balance of this paper is organized as follows. In the next section, we present an overview of SNP. This provides a quick introduction to SNP before delving into details in later sections. In Section 3, we elaborate on a list of design requirements and decisions we have made in the design of SNP. In Section 4, we provide a high-level description of the services offered by SNP. In Section 5, we give a specification of the SNP interface. In Section 6, we discuss various considerations that arise in implementing SNP. In Section 7, we provide some figures on the performance of our implementation. In Section 8, we compare SNP to some related systems. In Section 9, we discuss the lessons learned and directions for future work.

2 Overview of SNP

2.1 A Quick First Look

To give a quick introduction of what SNP is, we begin by looking at actual SNP code fragments. Figures 1 and 2 show respectively the typical client and server SNP code.

As can be easily seen, the SNP interface closely resembles that of sockets. This resemblance is not a coincidence. Rather, it was a design decision (see Section 3 for the rationale). In fact, most of the calls even retain their familiar semantics from their socket counterparts, though their implementations are quite different. In the following, we will focus only on the calls that are new in SNP.

There are two main new calls, namely `snp()` and `snp_attach()`. `snp()` replaces the `socket()` call in the socket interface. It is similar in functionality to `socket()` in that it creates a communication endpoint. It differs from `socket()`, however, in that an SNP endpoint corresponds to an application layer entity rather than a transport layer entity.

In addition to `SOCK_STREAM` and `SOCK_DGRAM`, `snp()` supports two new kinds of communication semantics: `SNP_STREAM` and `SNP_DGRAM`. Both extend the semantics of their respective socket counterparts by adding security guarantees. Specifically, an `SNP_STREAM` connection is authenticated. That is, a connection would be made only if it is accepted by the intended peer (specified by the initiator); and conversely, the identity of the initiator can be uniquely determined by the intended peer once a connection is made. Additional security services (e.g., data integrity, data confidentiality) can be activated on an `SNP_STREAM` connection by setting the appropriate options using `snp_setopt()` (see Section 4). Essentially, an `SNP_STREAM` connection can be understood as a connection that supports the semantics of a `SOCK_STREAM` connection even in an environment with intruders.[5] The case for `SNP_DGRAM` is similar.

`snp_attach()` is a completely new call; it does not have a socket counterpart. The main function of this call is to attach an *identity* to an SNP endpoint. The attached identity is the one that would be authenticated to a peer. An identity is not just a name, it is a *supported* claim of a particular name.[6] In other words, an identity can be unambiguously verified to another party. In terms of implementation, an identity consists of a name together with a set of *credentials* that corroborate the authenticity of the name. Typically, the operation of `snp_attach()` involves collecting the appropriate credentials (locally and/or remotely) for supporting the specified name. It should be noted that the identity attached to an SNP endpoint needs not be the identity of the caller *per se*. For example, a delegate may want to attach its identity as a delegate rather than its own identity. Operationally, a caller is allowed to attach any identity to an endpoint as long as it is able to gather the required credentials to support that identity.

[4] This is not strictly true as sockets also provide access to protocols in other layers in the communication hierarchy, cf., *raw sockets* etc. However, sockets and TLI are typically considered to be transport layer interfaces.

[5] Assuming proper options are set corresponding to the types of threats anticipated.

[6] A better term for this is *principal*. But we intend to keep it informal in this paper and refrain from introducing too many terms.

```
#include <snp.h>
        ...
        if ((snp_ep = snp(AF_INET, SNP_STREAM, SNP_PROTO_DEFAULT)) < 0) {
                snp_perror("snp() error :");            exit(1);
        }
        /* Initialize local and peer addr structs - just as in  sockets */
        ...
        /* Initialize local & peer name structs as shown below */
        local_name.np.np_val     = (char *) malloc(sizeof(client_name));
        local_name.np.np_len     = strlen(client_name);
        strcpy(local_name.np.np_val, client_name);
        peer_name.np.np_val      = (char *) malloc(sizeof(server_name));
        peer_name.np.np_len      = strlen(server_name);
        strcpy(peer_name.np.np_val, server_name);

        if (snp_attach(snp_ep, &local_name, &peer_name) < 0) {
                snp_perror("snp_attach() error :");     exit(1);
        }
        if (snp_connect(snp_ep, sizeof(struct sockaddr_in),
                        (struct sockaddr *) (&peer_addr) ) < 0) {
                snp_perror("snp_connect() error :");    exit(1);
        }
        if ((numbytes = snp_write(snp_ep, buf, buf_size)) < 0) {
                snp_perror("snp_write() error :");      exit(1);
        }
        ...
        if (snp_close(snp_ep) < 0) {
                snp_perror("snp_close() error :");      exit(1);
        }
        ...
```

Figure 1: Sample SNP Client Program Fragment

```
#include <snp.h>
        ...
        if ((snp_ep = snp(AF_INET, SNP_STREAM, SNP_PROTO_DEFAULT)) < 0) {
                snp_perror("snp() error :"); exit(1);
        }
        /* Initialize local and peer addr structs - just as in sockets */
        ...
        /* Initialize local name structs as shown below */
        local_name.np.np_val     = (char *) malloc(sizeof(server_name));
        local_name.np.np_len     = strlen(server_name);
        strcpy(local_name.np.np_val, server_name);
        if (snp_attach(snp_ep, &local_name, &peer_name) < 0) {
                snp_perror("snp_attach() error :");     exit(1);
        }
        if (snp_bind(snp_ep, &server_addr, sizeof(server_addr)) < 0) {
                snp_perror("snp_bind() error :");       exit(1);
        }
        if (snp_listen(snp_ep, 5) < 0) {
                snp_perror("snp_listen() error :");     exit(1);
        }
        if ((new_snp_ep = snp_accept(snp_ep, (struct sockaddr *) &peer_addr,
                                        &addr_len)) < 0) {
                snp_perror("snp_accept() error :");     exit(1);
        }
        if (snp_getpeerid (snp_ep, &client_name) < 0) {
                snp_perror("snp_getpeerid() error :");  exit(1);
        }
        if ((numbytes = snp_read(new_snp_ep, buf, buf_size)) < 0) {
                snp_perror("snp_read() error");         exit(1);
        }
        ...
        if (snp_close(new_snp_ep) < 0) {
                snp_perror("snp_close() error :");      exit(1);
        }
        ...
        if (snp_close(snp_ep) < 0) {
                snp_perror("snp_close() error :");      exit(1);
        }
        ...
```

Figure 2: Sample SNP Sequential Server Program Fragment

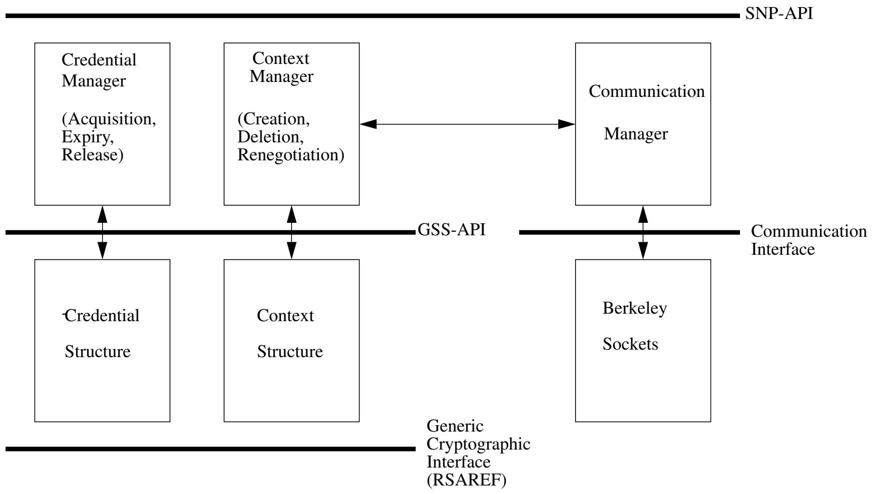

Figure 3: Organization of SNP

2.2 Components of SNP

SNP is designed and implemented in a modular fashion. Each major functionality of SNP is encapsulated in a separate layer that exports a well-defined interface. Figure 3 shows the conceptual layering and major components of SNP.

SNP-API defines the upper (external) interface available to an application. Internally, SNP makes use of two lower interfaces: GSS-API and a (insecure) network communication API. GSS-API encapsulates the details of the particular authentication protocol used, thus enhancing the independence of the SNP layer from the underlying authentication mechanism. Similarly, the communication API isolates the details of network communication from the SNP layer. We have chosen sockets as the communication API, mainly due to its wide availability.

GSS-API in turns makes use of a lower generic cryptographic interface. This interface provides access to all cryptographic functions and is generic in the sense that it can support any (symmetric or asymmetric) cryptosystem. This provides "cryptosystem independence" and facilitates easy substitution when new (implementations of) cryptosystems are available. Further discussion of our GSS-API implementation and the underlying authentication protocol is beyond the scope of this paper; interested readers can consult [23, 20] for more details.

The main function of the SNP layer is *context* and *credential management*. It initiates the acquisition of credentials, monitors the status of contexts and credentials, and initiates renegotiation (of contexts) and/or reacquisition (of credentials), if necessary. It should be noted, however, that the actual storage of contexts and credentials is internal to GSS-API.

2.3 SNP in Context

SNP is part of a larger project of ours that concerns the design and implementation of an *authentication framework* for distributed systems [21]. The framework addresses a range of authentication needs that includes bootstrapping, user logins and peer communications. SNP is designed as an interface for accessing the peer authentication protocol in our framework.

Because of its modular design, detailed understanding of the other components in the framework is not required in order to use or understand SNP. Indeed, SNP is relatively independent of the original framework it was designed for, and should be easily portable for use in other authentication frameworks (see Section 3). Therefore, we will only briefly describe the other components in our framework, to the extent they are required for the understanding of SNP.

At present, our framework has three protocols: a *secure bootstrap protocol* that creates a *bootstrap certificate* upon successful bootstrapping; a *user-host mutual authentication protocol* that creates a *login certificate* when a user successful logs in; and a *peer-peer mutual authentication protocol* that is the basis for SNP. The login certificate is retrieved when a user *attaches* its identity to an SNP endpoint. This certificate is stored in an SNP (GSS-API) credential structure and is used to authenticate the user's identity to its peer.

The peer-peer authentication protocol in our framework assumes the use of a commonly trusted *authentication server* (AS). Apart from its authentication duty, AS is also responsible for generating the session key used in an authentication exchange. Thus, in order for SNP to function correctly, AS needs to be properly set up. For example, it must be properly secured and be given a correct database. The discussion of the associated administrative issues is beyond the scope of this paper.

Lastly, a *name service* is required for translating application layer entities to their transport layer addresses. This name service, however, need not be trusted, as SNP performs the proper authentication during connection establishment.

3 Design Requirements and Decisions

In designing SNP, we first set out a number of requirements. Based on these requirements, we made several key design decisions. We briefly discuss the rationale for these requirements and decisions below:

- SNP should provide end-to-end communication at the application layer rather than the transport layer. Although the transport layer is the first end-to-end layer, we believe the concept of identity is only meaningful at or above the session layer. For example, in Unix and TCP/IP, ports are ephemeral and the association of ports with processes is dynamic.[7] We believe it is more appropriate to base our semantics on application level entities than to assume a secure mapping between ports and processes.

- SNP should be independent of any particular authentication protocol or framework. This allows SNP to be portable across different authentication systems. We achieve such independence by using GSS-API to encapsulate the details of the underlying protocol, and sockets as the communication interface.

- It should be easy to convert existing network application programs to use SNP. To achieve this, we designed SNP to retain as much as possible the general structure of a socket program. Hence: (1) only a minimal number of new concepts needs to be learned in order to acquaint oneself with SNP; (2) only minor (mostly syntactic) modifications need to be done to convert a socket program to an SNP program, thus significantly facilitating retrofitting.

  We could have emulated the TLI interface instead. But we believe that sockets and TLI are sufficiently similar to each other that little extra effort is required to convert TLI programs into SNP programs. Besides, there are far more existing socket programs than TLI programs, though TLI is quickly gaining popularity.

- SNP should work in a heterogeneous environment. This entails careful considerations of message encoding and processing. We have chosen XDR [16] for this purpose, mainly for its simplicity. ASN.1 [1] is used in other authentication systems (e.g., Kerberos, SPX); we find it to be overly complicated and not suitable as a prototyping tool. From our experience, XDR has been adequate, though not as flexible as we would like.

- SNP should be independent of particular cryptosystems. We achieve this by encapsulating all cryptographic functions using a generic cryptographic interface. In our current implementation, we use the *de facto* standard cryptosystem trio, i.e., DES [10] for symmetric encryption, RSA [13] for asymmetric encryption and MD5 [12] for message digest.

[7] Reserved ports are a matter of convention only, there is no permanent binding.

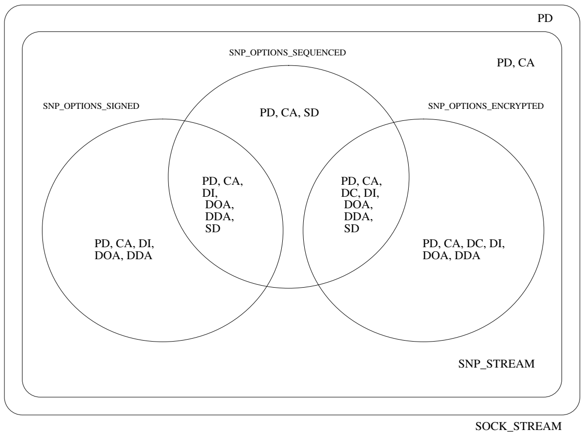

Figure 4: Services Provided

4 Services Provided

Security is only well-defined with respect to a threat model. In this paper, we assume the standard threat model. That is, a saboteur can read, insert, delete and modify any network traffic. It should be noted that a saboteur is not necessarily a totally external intruder, s/he can also be a legitimate user. Thus, s/he can use information available to a legitimate user in mounting an attack.

We stress that our model does not include *denial of service* and *traffic analysis* threats. It is always possible for a saboteur to corrupt all packets passing through. Even an infinitely persistent sender cannot overcome such corruption if the saboteur is equally persistent. Indeed, most network programming abstractions guarantee only *safety* but not *progress*.

In the following, we present in high-level terms the services provided by SNP. We first define below the typical types of services offered by a secure communication connection:

- *Persistent Delivery* (PD) — A sender will persistently try to retransmit data if it has not been received yet.

Thus, PD implicitly assumes the use of acknowledgments.

- *Best Effort Delivery* (BED) — Data sent may or may not arrive at the receiver. Each of the intermediate nodes can either forward or drop the data.
- *Sequenced Delivery* (SD) — If data arrives at a receiver, it must appear in the same order it was sent. That is, no reordering or duplication is allowed.
- *Data Confidentiality* (DC) — Data is only legible to the intended receiver.
- *Data Integrity* (DI) — Data, if accepted by a receiver, must bear the same content as that sent.
- *Data Origin Authenticity* (DOA) — Data, if accepted by a receiver, must have come from a known specific sender.
- *Data Destination Authenticity* (DDA) — When data arrives, a receiver can unambiguously determine that it is the intended receiver.
- *Connection Authenticity* (CA) — A connection, if made, must be between the intended peers.

SNP can provide different combinations of the above services. The precise combinations provided is summarized in Figure 4. Each of the two boxes is labeled by a constant denoting the communication semantics, while each of the circles is labeled by an SNP option constant. The combination of services provided under a particular communication semantics and set of options is labeled in the intersection of the corresponding regions. For example, under `SNP_STREAM`, if both `SNP_OPTIONS_SIGNED` and `SNP_OPTIONS_SEQUENCED` are set, the services provided are PD, CA, DI, DOA, DDA and SD. We note that `SNP_OPTIONS_SIGNED` and `SNP_OPTIONS_ENCRYPTED` cannot both be set at the same time. The case for `SNP_DGRAM` is similar, and is omitted.

5 The SNP Interface

As with the socket interface, SNP-API functions can be divided into five classes: initialization, connection establishment, data transfer, connection release, and utility. We describe the functions in each class below. A complete list of all functions is given in Figure 5. Parameter names appearing in the following subsections refer to those shown there.

Most functions have semantics similar to their socket counterpart. (In fact, they are given the same names modulo the prefix "`snp_`.") We have not emulated all the data transfer functions of sockets (e.g., `sendmsg`, `recvmsg`) due to their intricate semantics. Nonblocking I/O is supported, but asynchronous I/O (i.e., interrupt driven) is not.

We also note that most functions below (notable exceptions being the data transfer functions) return 0 on success and -1 on failure. In addition, a global variable `snp_errno` will contain the appropriate error number on failure.

5.1 Initialization

Functions in this class are used for creating and initializing an SNP endpoint. They include `snp()`, `snp_bind()`, `snp_listen()` and `snp_attach()`.

5.1.1 `snp()`

`snp()` creates an endpoint of communication. Its parameters have the same types as `socket()` and have similar semantics. Currently, the only supported value for `family` is `AF_INET`, corresponding to the internet address family. The possible values of `type` are shown in the following table:

`SNP_STREAM`	Secure Stream
`SNP_DGRAM`	Secure Datagram
`SOCK_STREAM`	Normal (Insecure) Stream
`SOCK_DGRAM`	Normal (Insecure) Datagram

For `protocol`, the currently supported values are as follows:

`SNP_PROTO_DEFAULT`	Default Authentication Protocol
`SNP_PROTO_PUSH_MODEL`	Push Model Authentication Protocol
`SNP_PROTO_REVERSE`	Reverse Authentication Protocol
`IPPROTO_TCP`	Normal TCP
`IPPROTO_UDP`	Normal UDP

A combination of `SNP_STREAM` and any one of the first three protocol values results in a secure equivalent of TCP. Similarly, `SNP_DGRAM` in combination with one of the first three protocol constants provides a secure UDP protocol. The first three protocol constants can be used only when the `family` argument value has been set to either `SNP_STREAM` or `SNP_DGRAM`. The use of either `IPPROTO_UDP` or `IPPROTO_TCP` results in the normal (i.e., insecure) UDP or TCP protocols, respectively. These are equivalent to the semantics provided by the socket interface.

`snp()` returns an SNP *handle*, of type `int`. The handle is an index into an internal table of SNP structures maintained by SNP. Thus, unlike `socket()`, an SNP handle is *not* a file descriptor. Hence, some of the standard functions that apply to a socket descriptor will not apply to an SNP handle.

The `snp_ep` parameter in each of the other functions in Figure 5 refer to an SNP handle obtained from a call to `snp()`.

Initialization Calls

```
int snp         ( int family, int type, int protocol );
int snp_bind    ( int snp_ep, struct sockaddr *local_addr, int addr_len );
int snp_listen  ( int snp_ep, int backlog );
int snp_attach  ( int snp_ep, struct name_s *local_name, struct name_s *peer_name );
```

Connection Establishment Calls

```
int snp_connect ( int snp_ep, struct sockaddr *peer_addr, int peer_addr_len );
int snp_accept  ( int snp_ep, struct sockaddr *peer_addr, int peer_addr_len );
```

Data Transfer Calls

```
int snp_write   ( int snp_ep, char *buf, int nbytes );
int snp_read    ( int snp_ep, char *buf, int nbytes );
int snp_send    ( int snp_ep, char *buf, int nbytes, int flags );
int snp_recv    ( int snp_ep, char *buf, int nbytes, int flags );
int snp_sendto  ( int snp_ep, char *buf, int nbytes, int flags,
                  struct sockaddr *to,   int tolen );
int snp_recvfrom (int snp_ep, char *buf, int nbytes, int flags,
                  struct sockaddr *from, int *fromlen );
```

Connection Release Calls

```
int snp_close   ( int snp_ep );
int snp_shutdown (int snp_ep, int how );
```

Utility Calls

```
int snp_setopt  ( int snp_ep, int level, int optname, char *optval, int optlen );
int snp_getpeerid ( int snp_ep, struct name_s *peer_name );
```

Figure 5: SNP Interface Specification

5.1.2 snp_bind()

After creation, an address may be bound to an SNP endpoint using snp_bind(). The local_addr and addr_len are of the same types as in the bind() function. They specify the address to be bound.

5.1.3 snp_attach()

snp_attach() is used for specifying the identity a caller wishes to be authenticated as to its peer and the name of the intended peer. The name structure name_s is of the following form: (This structure is automatically generated by rpcgen from a XDR structure.)

```
struct name_s {
  struct {
    u_int np_len;   /* Length of the name */
    char  *np_val;  /* The actual name    */
  } np;
};
```

If invoked by a server, peer_name may be set to NULL, in which case connection from any client would be accepted. Once a connection is established, the identity of the client can be discovered by calling snp_getpeerid() (see below). snp_attach() *must* be invoked before connection establishment, if secure communication is desired.

5.1.4 snp_listen()

The function allows its caller to specify the maximum allowed backlog of connection requests. It has identical semantics as listen(), except it takes an SNP handle. Typically, a caller of snp_listen() is a server. This function can only be used on an SNP_STREAM or SOCK_STREAM connection.

5.2 Connection Establishment

The second class of functions consists of snp_connect() and snp_accept(); they are mostly used for stream connections.

5.2.1 snp_connect()

For an SNP_STREAM endpoint, this function results in the establishment of a connection with a peer if a corresponding snp_accept() is performed by the peer. A successful connection also indicates a successful authentication exchange using the underlying authentication protocol.

In the case of SNP_DGRAM, snp_connect() only saves the supplied peer address in an internal SNP structure. This address would be assumed to be the destination address in all subsequent data transfer unless an explicit address is given. No authentication is performed at the time of the call; instead, it is performed at the time of the first data transfer call.

5.2.2 snp_accept()

snp_accept() can be used only on an SNP_STREAM or SOCK_STREAM endpoint. It accepts connection requests and completes them if the authenticated peer identity matches the one specified by a previous snp_attach().[8] Successful completion also implies that the peer identity has been authenticated, and can be discovered using snp_getpeerid(). Furthermore, it implies the establishment of a pair of security contexts (one at each peer) and the distribution of a session key.

The return value is a new SNP handle which can be used for further communication with the peer. Further connection requests can continue to come in on the original SNP endpoint. If peer_addr and peer_addr_len are non-NULL, they will be filled in appropriately.

5.3 Data Transfer

All of the following data transfer functions return the number of bytes actually sent or received on success and -1 on failure.

5.3.1 snp_sendto()

snp_sendto() sends nbytes of data pointed to by buf to the peer address specified by the to parameter. This function may be used on both stream and datagram endpoints. In case of a datagram endpoint, both to and to_len must be specified. The data will be sent encrypted or signed if the appropriate SNP options have been set (see snp_setopt() below). The possible values and semantics of flags are the same as those in sendto().

5.3.2 snp_recvfrom()

snp_recvfrom() attempts to receive nbytes of data and stores them in a buffer pointed to by buf. The address and address length of the peer are filled into from and from_len respectively, if both of them are non-NULL. flags has the same semantics as in the recvfrom(). The incoming data is decrypted or verified, depending upon the SNP options specified.

5.3.3 snp_read(), snp_write(), snp_send() and snp_recv()

These calls can only be used on stream endpoints. Their semantics are essentially similar to their socket counterparts. snp_send() and snp_recv() provides additional features (e.g., such as expedited data) that are not available with snp_write() and snp_read(). The nature of data sent or received depends on the current SNP options.

[8] If the peer name specified is NULL, connections from any client is accepted.

5.4 Connection Release

5.4.1 snp_shutdown() and snp_close()

These functions have similar semantics as their socket counterparts, except they perform the release only after they have verified that the release request did originate from the correct peer.

5.5 Utility Routines

These functions are used for manipulating or retrieving the characteristics of an SNP endpoint.

5.5.1 snp_setopt()

snp_setopt() is used to set options available for a regular socket as well as those specific to SNP. A new constant, SNP, has been introduced for the level parameter. The options available at the SNP level are:

SNP_OPTIONS_DEFAULT	Reset all option settings to default
SNP_OPTIONS_ENCRYPTED	Encrypt all subsequent data
SNP_OPTIONS_SIGNED	Sign all subsequent data
SNP_OPTIONS_SEQUENCED	Enforce sequencing on data
SNP_OPTIONS_NOTIFY	Notify caller on context expiry — do not reinitiate authentication
SNP_OPTIONS_CONTEXT_TIME	Set context expiration time

Setting SNP_OPTIONS_DEFAULT results in resetting all options to their default settings; that is, no encryption, no signing and no sequencing.

Setting SNP_OPTIONS_ENCRYPTED causes subsequent outgoing data to be encrypted. Setting SNP_OPTIONS_SIGNED causes subsequent outgoing data to be signed. The key to be used for encryption and signing is the session key maintained in the current security context. Options SNP_OPTIONS_ENCRYPTED and SNP_OPTIONS_SIGNED cannot be set at the same time. To enforce sequencing of data, option SNP_OPTIONS_SEQUENCED should be set. This may be used in conjunction with either SNP_OPTIONS_ENCRYPTED or SNP_OPTIONS_SIGNED.

When the current security context expires, the SNP layer automatically renegotiates a new context. This can be disabled by setting SNP_OPTIONS_NOTIFY; in which case, the SNP user will be notified of context expiry when it performs an SNP call. The duration of a context can be set using the SNP_OPTIONS_CONTEXT_TIME option.

Note that the first five options are toggle flags, while the last one requires the context duration to be specified in optval.

5.5.2 snp_perror() and snp_getpeerid()

snp_perror() performs the same function as the standard perror() function, except that it accounts for

Connection Establishment

(CE1)	I	:	generate nonce n_I
(CE2)	$I \rightarrow R$	:	I, n_I
(CE3)	R	:	generate nonce n_R
(CE4)	$R \rightarrow AS$	:	I, n_I, R, n_R
(CE5)	AS	:	generate key k
(CE6)	$AS \rightarrow R$	:	$\{\{I, n_I, R, n_R, k\}_{k_{AS}^{-1}}\}_{k_R}$
(CE7)	$R \rightarrow I$	:	$\{\{I, n_I, R, n_R, k\}_{k_{AS}^{-1}}\}_{k_I}, \{n_I, n_R\}_k$
(CE8)	$I \rightarrow R$	:	$\{n_R\}_k$

Connection Release

(CR1)	$I \rightarrow R$	:	$\{I, n_I, R, n_R\}_k$
(CR2)	$R \rightarrow I$	:	$\{n_I, n_R\}_k$

Figure 6: Underlying Authentication Protocol

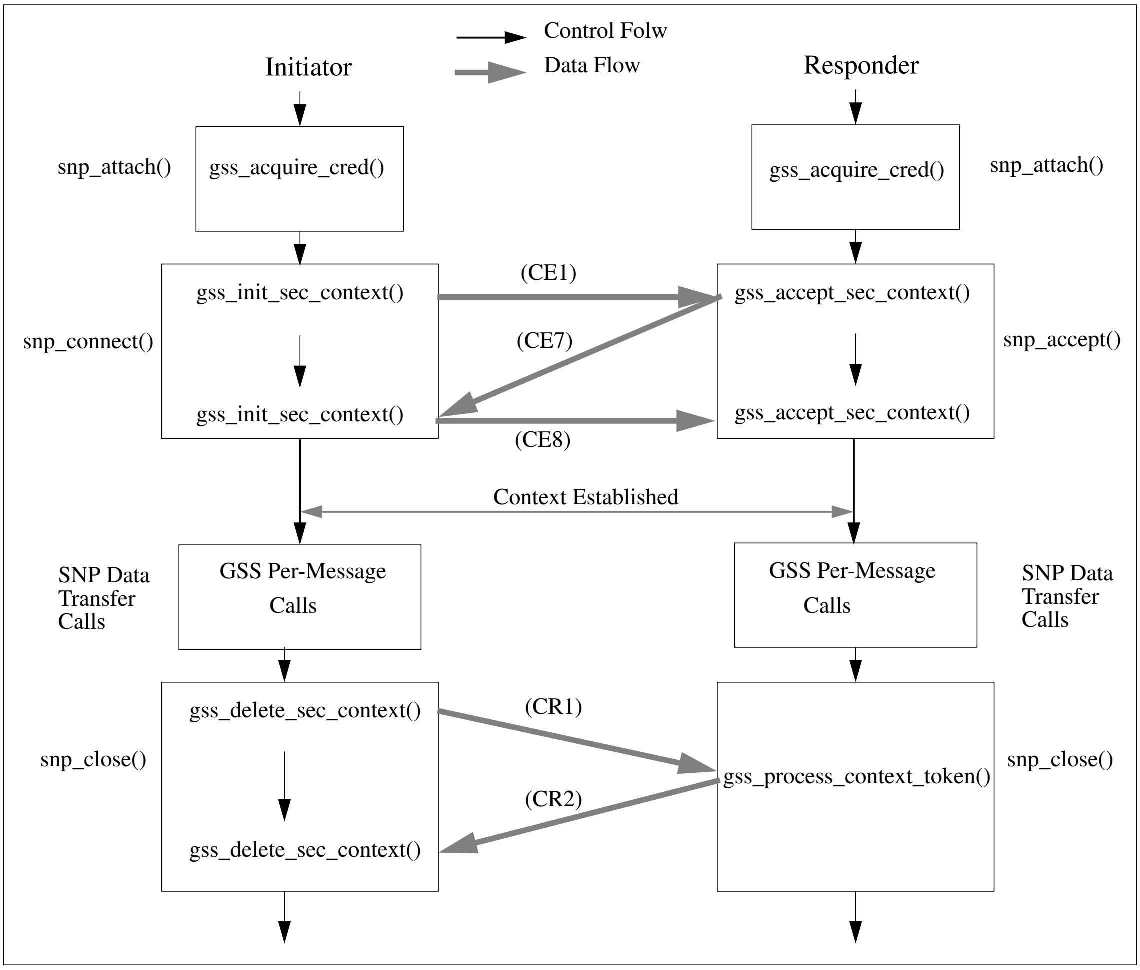

Figure 7: Control and Data Flow

SNP-API error codes as well. `snp_getpeerid()` retrieves the authenticated identity of the peer.[9]

[9] In fact, the unauthenticated identity of the peer is available as soon as the underlying authentication protocol has proceeded beyond a certain point, even if the authentication exchange fails at the end.

6 Overview of Implementation

To facilitate discussion of SNP's implementation, it is helpful to first briefly describe our implementation of GSS-API. The authentication protocol underlying our GSS-API implementation is shown in Figure 6 (I denotes the *initiator*, R the *responder* and AS the authentication server). The protocol was initially published in [22], and later verified in [20, 23]. The mapping of this protocol to GSS-API is quite straightforward, and is described in [23]. The key point to note is that the communications with AS (steps (CE4)–(CE6)) are completely encapsulated within GSS-API, and are not observable by the SNP layer.

Typically, an SNP-API call is translated into a number of GSS-API calls together with calls to the communication layer. GSS-API is responsible for generating *tokens* that are to be shipped using the communication layer. In simple terms, the main responsibility of the SNP layer is to request the right tokens to be generated (according to user request and current state) and to ensure they are properly transferred to the peer SNP layer. Figure 7 shows the relationship between SNP-API calls and GSS-API calls. (The bold arrows in Figure 7 correspond to the protocol steps in Figure 6.) For example, a call to `snp_connect()` results in two calls to `gss_init_sec_context()` as well as three calls to the communication layer.

There are several major considerations in implementing SNP. We describe them below:

- Two types of messages, namely, *data* and *control*, are transferred between SNP peers. Data messages contain user data and correspond to SNP data transfer calls, while control messages contain information related to the operation of the SNP layer (e.g., connection establishment request/response) and correspond to SNP control calls (e.g., `snp_connect()`) and functions (e.g., context renegotiation).

  There are two ways these messages can be transferred. One is to multiplex them onto a single connection, and the other is to create dedicated connections for each type of messages. We opted for the latter because control messages should generally be given priority over data messages. Thus, if they are to be transferred on the same connection, the underlying communication mechanism must support some form of *priority* message facility. Most existing communication mechanisms (sockets in particular) do not support such priority message processing well.[10] The two-connections solution avoids the dependence on such a mechanism.[11]

- The two-connections solution also simplifies buffering concerns. Specifically, by always reading from the control connection (and responding to it) first, we no longer need to buffer all the user data preceding a control message if a control action is needed. The elimination of extra buffering also improves performance.

- The use of two connections raises the question of the address to which the second connection should be bound. Our current implementation always establishes the second (i.e., control) connection at a fixed offset from the user supplied (i.e., data) connection address. If adopted as a convention, this should not create any collision problem.

The main data structure in the SNP layer is the `snp_struct` structure. Its definition is shown in Figure 9. The `control_sockfd` and `data_sockfd` fields contain, respectively, the socket descriptors for the control and data connections. The fields `cred_list_ptr` and `ctx_list_ptr` contain pointers to GSS layer structures (see Figure 8). The meanings of most other fields are given in the comments. Each call to `snp()` creates an `snp_struct` structure; the SNP handle returned is an index into an internal table of pointers to `snp_struct` maintained by the SNP layer.

We have only touched upon the main ideas in our implementation. Most of the details concerning context expiration, context renegotiation, etc., have been omitted due to length limitation. This paper is intended only as a preliminary overview. We hope to provide a full account in a final report.

7 Performance

In this section, we present some performance results of our SNP implementation. The measurements were done on a network of Sun SPARCstations 10/30 running SunOS 4.1.3. The resolution of the system clock is in the order of microseconds.[12]

We first calibrate the performance of our cryptographic packages. Our DES package is a generic public domain one, while our RSA/MD5 package is from RSAREF [4]. Both packages are relatively portable, and are not optimized. The calibration allows us to determine the overhead introduced by the SNP layer, excluding cryptographic cost. This provides a better measure of the performance of our SNP implementation, because as more highly optimized cryptographic packages and hardware become available, the cryptographic cost will diminish, while the SNP overhead remains constant.

[10] TCP does not support *out-of-band* data. It does support some elementary form of urgent data with the *urgent bit* and the *urgent pointer*. Berkeley socket supports out-of-band data, though the precise semantic guarantee is highly implementation-dependent.

[11] In some sense, this is arguable because typically, there is no guarantee on the relative arrival times of messages sent on different connections. However, in practice, for connections with the same source and destination, the times of arrival closely follow the times of the respective sends.

[12] The measurement error, however, is much worse because of context switching, function call overhead, etc.

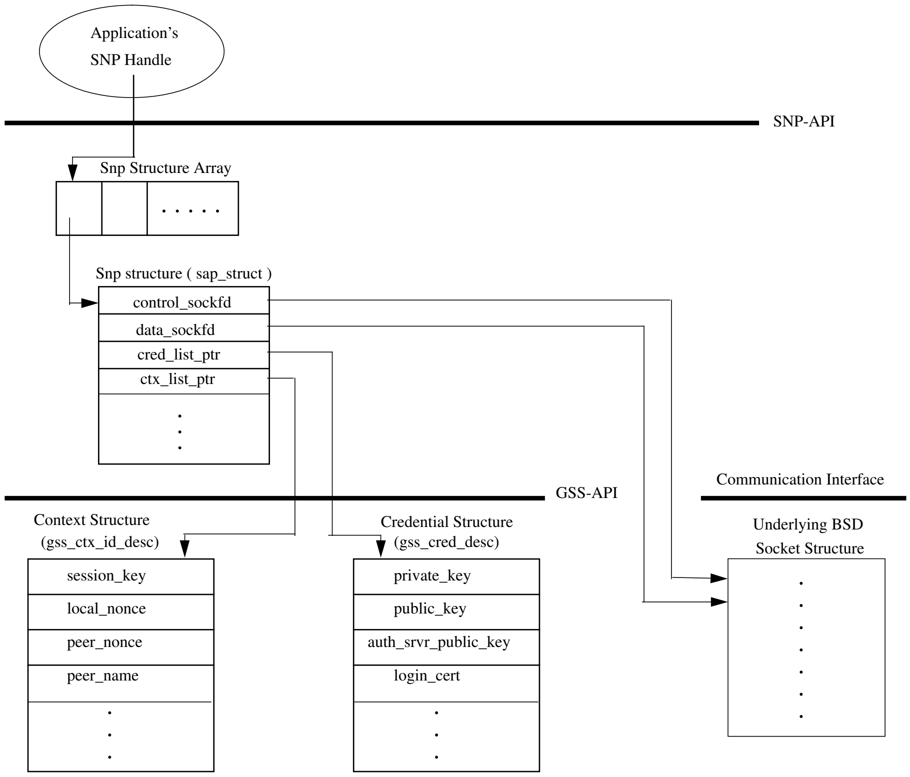

Figure 8: Data Structures

```
struct snp_struct {
        int                     control_sockfd;      /* Control socket desc         */
        int                     data_sockfd;         /* Data    socket desc         */
        int                     family;              /* Params specified in call    */
        int                     type;
        int                     protocol;
        struct sockaddr         *local_addr;         /* Obtained from snp_bind()    */
        int                     local_addr_len;
        struct sockaddr         *peer_addr;          /* Obtained from snp_connect   */
        int                     peer_addr_len;       /* or first data xfer calls    */
        struct name_s           *local_name;         /* Obtained from snp_attach()  */
        struct name_s           *peer_name;
        gss_cred_id_t           cred_list_ptr;       /* Credentials pointer         */
        gss_ctx_id_t            ctx_list_ptr;        /* Context pointer             */
        int                     secure_options;      /* Obtained from snp_setopt()  */
        int                     no_send;             /* Options for snp_shutdown()  */
        int                     no_recv;
        struct msg_s            *remaining_data;     /* Data recd but not requested */
        a_uint16                seq_number;          /* For the GSS sequencing      */
        a_uint16                recd_seq_number;
}
```

Figure 9: SNP Structure

Data Length	16B	512B	1KB	2KB	4KB	8KB	16KB	32KB
DES Encryption	0.42	2.85	5.25	9.97	19.15	37.47	75.19	152.58
DES Decryption	0.41	2.94	5.40	10.19	19.54	38.20	77.24	158.78
DES Sign	0.36	0.54	0.73	1.14	1.89	3.46	6.61	12.72
DES Verify	0.33	0.54	0.73	1.11	1.89	3.42	6.57	12.75
RSA 512 Encryption	541.24	542.21	546.00	551.92	560.44	577.69	618.54	689.76
RSA 512 Decryption	53.93	56.85	59.11	63.83	73.14	91.29	127.49	198.42
RSA 512 Sign	540.25	540.10	540.45	540.64	544.82	544.01	550.25	551.17
RSA 512 Verify	53.86	53.82	54.20	54.49	55.48	57.06	60.10	66.21
MD5	0.056	0.25	0.43	0.79	1.52	3.00	6.00	12.11

Table 1: Cryptographic Performance (in milliseconds)

		Connect		Close	
		Number of Operations	Subtotal	Number of Operations	Subtotal
RSA	Encryption	2@1kB	1092.00		
	Decryption	2@1kB	118.22		
	Sign	1@1kB	540.45		
	Verify	2@1kB	108.40		
DES	Encryption	2@200B	2.85	2@200B	2.85
	Decryption	1@200B	1.45	2@200B	2.90
	Key Gen.	1	0.38		
XDR	Encode	21@600B	29.82	2@200B	2.84
	Decode	20@600B	2.00	2@200B	0.20
Total Time			1895.57		8.79
Measured Time			2148.40		24.90
SNP Overhead			252.83		16.11
Socket Time			2.40		0.40

Table 2: Connect and Close Calls (in milliseconds)

Buffer Length	1kB	2kB	4kB	8kB	16kB
Socket SOCK_STREAM	0.6	1.0	1.3	2.8	5.2
SNP SOCK_STREAM	0.7	1.1	1.5	3.4	6.7
SNP Overhead	0.1	0.1	0.2	0.6	1.5
SNP_STREAM Plain	2.2	3.2	4.4	7.1	12.6
SNP_STREAM Signed	4.2	5.8	8.8	14.6	27.1
SNP_STREAM Encrypted	13.0	22.9	42.7	82.8	163.6

Table 3: Data Transfer Calls (in milliseconds)

Referring to Table 1, the following observations can be made: (1) The performance of both DES (CBC mode) and MD5 is linear with respect to data size. (2) The performance of RSA is also linear except for small data sizes. This is due to the fact that for large data sizes, the RSA implementation does not perform "true" RSA encryption. Instead, it first generates a random DES key, then encrypts the data with the DES key, and finally encrypts the DES key using RSA.

Our measurements of SNP performance are given in Tables 2 and 3. All measurements are for `SNP_STREAM`; similar measurements apply to `SNP_DGRAM`, and are omitted. Note also that these measurements are based on the use of 512-bit RSA keys (i.e., modulus).[13]

Table 2 shows the timing results for connection establishment (i.e., `snp_connect()`/`snp_accept()`) and release (i.e., `snp_close()`). The Total Time row gives the amount of time accounted for by cryptographic and XDR operations. The Measured Time row gives the observed times in establishing and closing an SNP connection. The difference between Measured Time and Total Time (the SNP Overhead row) gives the overhead introduced by SNP. The Regular Socket row gives the time it takes for the corresponding socket calls to complete. Thus, for connection establishment, SNP introduced around 0.2s overhead. A major component of this overhead is the extra round-trip delay for the communication with the authentication server and the associated message processing at the authentication server. For connection release, the SNP overhead is around 16ms.

Table 3 shows the timing results for data transfer calls (specifically for `snp_write()`). The first two rows give the times for a `SNP_STREAM` connection with the encrypt and sign options set, respectively. The third row gives the time for a regular `SNP_STREAM` with no option set, whereas the fourth row gives the time using plain sockets. The SNP Overhead row gives the overhead introduced by SNP. It can be observed that the SNP overhead is minimal.

[13] Our implementation is parametric with respect to key length. We can easily switch over to 1024-bit keys. That, however, will slow things down significantly. The increase in cost is not linear in key length.

Two conclusions can be drawn from these measurements: (1) The cost of cryptographic operations dominates the total cost of SNP. We believe this can be generalized to any cryptographic security mechanism. (2) It is possible to provide security at the application layer without incurring undue overhead, even with an unoptimized implementation. We expect a streamlined implementation to perform even better.

8 Related Work

Most existing work on secure network communication is focused on the protocol or architecture aspects [3, 9, 15, 17]; not much has been done concerning a general secure application network programming interface.

The work most relevant to ours includes several secure RPC systems: the secure RPC package in [2], Sun secure RPC [18] and DCE secure RPC [14]. The goals of these systems are similar to ours: to provide applications transparent access to secure communication. However, the models of communication adopted are different. RPC assumes an *implicit* communication model. That is, its users do not directly manage communications, but instead they deal with high-level abstractions in terms of procedures. SNP assumes an *explicit* communication model; SNP users are directly responsible for initiating connections, sending and receiving data, and closing connections. The same difference exists between sockets/TLI and RPC styles of network programming.

Apart from this, the implementation of these RPC systems is totally different from ours. For example, they tend to be tightly coupled to the underlying protocol (e.g., a modified Needham-Schroeder protocol [11] is used in [2], Kerberos is used in DCE). Our use of GSS-API provides protocol independence.

A recent paper by Wobber *et al.* [19] describes an operating system interface for supporting authentication. The interface is based on a formal theory of a *speaks for* relation [7]. Its concrete implementation contains several interesting abstract datatypes, e.g., a `Prin` type that represents *principals*, and an `Auth` type that represents principals a process can speak for. In relating to our work, their interface can be used as an alternate lower interface for SNP. In other words, instead of translating SNP-API calls to GSS-API calls, they can be translated to calls to the interface in [19]. Such a translation should be quite straightforward because of the high level of abstraction supported. A major disadvantage of their interface, though, is the lack of compatibility with other security mechanisms, e.g., Kerberos. Moreover, their interface has only been implemented on the Taos operating system, and is currently not available on Unix.

9 Discussion and Future Work

We believe SNP represents an important first step toward secure network programming for the masses. It is clear that many important issues need to be resolved before this could be a reality. Some of these issues are: the development of a *security infrastructure* that provides uniform management and distribution of credentials (particularly for interdomain authentication), and operating system support for basic security concepts such as identity (see [19]).

One of the other impediments is performance. With rapidly improving cryptographic software and hardware, this should be a diminishing problem. As demonstrated in [5], the speed of a modern RISC-based workstation is already quite adequate for most cryptographic computation, provided the right algorithms and optimizations are used.

We are also considering several interesting extensions to the SNP interface. First, delegation can be added. This would involve the addition of two new calls: `snp_delegate()` for the delegating process and `snp_assume()` for the delegate. Delegation allows a delegate to act with the same authority as the delegating process. Second, the `snp_attach()` call can be extended to accept *identity expressions* instead of just simple identity specifications. An identity expression can specify a combination of identities that would be communicated to the peer.

In terms of implementation, we may try to port SNP to other authentication systems conforming to GSS-API. Also, the essential ideas of SNP can be adapted to provide security at other layers (e.g., transport). The lessons we learned in designing and implementing SNP provide useful references in such an effort.

Concerning the design of our interface, we have made the compatibility with sockets as one of our top design requirements. With our present design, a typical socket program can be converted into an SNP program by simply adding an `snp_attach()` call,[14] without significantly modifying any of the existing code. Alternate designs with less compatibility are possible. For example, the concept of identity can be promoted to "first class citizen" status, replacing completely the use of socket addresses. For example, the functions `snp_connect()` and `snp_accept()` would then become

```
int snp_connect ( int snp_ep,
        struct name_s *peer_name );
int snp_accept  ( int snp_ep,
        struct name_s *peer_name );
```

Another concern in the interface design is *user control*. How much control should a user be given and how should it be done? For example, users (with the help of an operating system) may wish to explicitly manage credentials themselves, or to use their own encryption keys or algorithms. Our present design allows very limited user control (mainly through `snp_setopt()`); this could be appropriately extended.

Finally, there is the question of what the best layer for providing security support for network communication is. It can be argued that there is no single best layer for this purpose. The question then becomes: what is the best placement of security functionalities into different layers so that the resulting architecture is most general and admits least duplication? Much more research is needed to obtain an answer.

[14] And also prefixing socket calls with "`snp_`".

Acknowledgments

We wish to thank our shepherd Adam Moskowtiz and the anonymous referees for their constructive comments and suggestions. We are also grateful to Dinesh Das and members of the Network Research Seminar of the University of Texas at Austin for listening to our ideas.

References

[1] CCITT Recommendation X.208 Specification of Abstract Syntax Notation one (ASN.1), 1988. See also ISO/IEC 8824, 1989.

[2] A.D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[3] J. Ioannidis and M. Blaze. The architecture and implementation of network-layer security under unix. In *Proceedings of 4th Usenix Unix Security Workshop*, Santa Clara, California, October 4–6 1993.

[4] RSA Laboratories. RSAREF: A cryptographic toolkit for privacy-enhanced mail. January 5 1993.

[5] J.B. Lacy, D.P. Mitchell, and W.M. Schell. Cryptolib: Cryptography in software. In *Proceedings of Usenix Unix Security Workshop IV*, pages 1–17, Santa Clara, California, October 4–6 1993.

[6] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 165–182, Asilomar Conference Center, Pacific Grove, California, October 13–16 1991.

[7] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992. A preliminary version of this paper appeared as [6].

[8] J. Linn. *Generic Security Service Application Program Interface*, September 1993. RFC 1508.

[9] R. Molva, G. Tsudik, E. Van Herreweghen, and S. Zatti. *KryptoKnight* authentication and key distribution system. In *Proceedings of 2nd European Symposium on Research in Computer Security*, pages 155–174, Toulouse, France, November 23–25 1992. Springer Verlag.

[10] National Bureau of Standards, Washingtion, D.C. *Data Encryption Standarad FIPS Pub 46*, January 15 1977.

[11] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[12] R. Rivest. *The MD5 Message-Digest Algorithm*, April 1992. RFC 1321.

[13] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[14] W. Rosenberry, D. Kenny, and G. Fisher. *Understanding DCE*. O'Reilley & Associates, Inc., 1992.

[15] J.G. Steiner, C. Neuman, and J.I. Schiller. *Kerberos*: An authentication service for open network systems. In *Proceedings of USENIX Winter Conference*, pages 191–202, Dallas, TX, February 1988.

[16] Sun Microsystems, Inc. *XDR: External Data Representation Standard*, June 1987. RFC 1057.

[17] J.J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of 12th IEEE Symposium on Research in Security and Privacy*, pages 232–244, Oakland, California, May 20–22 1991.

[18] B. Taylor and D. Goldberg. Secure networking in the Sun environment. In *Proceedings of Summer Usenix Conference*, pages 28–37, Atlanta, Georgia, June 1986.

[19] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. In *Proceedings of 14th ACM Symposium on Operating Systems Principles*, Ashville, North Carolina, 1993.

[20] T.Y.C. Woo. *Authentication and Authorization in Distributed Systems*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, May 1994.

[21] T.Y.C. Woo and S.S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, January 1992.

[22] T.Y.C. Woo and S.S. Lam. "Authentication" revisited. *Computer*, 25(3):10, March 1992.

[23] T.Y.C. Woo and S.S. Lam. Design, verification, and implementation of an authentication protocol. Technical Report TR 93-31, Department of Computer Sciences, The University of Texas at Austin, November 1993.

Thomas Y.C. Woo is a Ph.D. candidate at the Department of Computer Sciences at the University of Texas at Austin. His research interests include computer networking, distributed systems, security and multimedia.

Thomas received a BS (First-Class Honors) degree in computer science from the University of Hong Kong and an MS degree in computer science from the University of Texas at Austin.

Raghuram Bindignavle is a Masters candidate at the Department of Computer Sciences at the University of Texas at Austin. He received his BE in computer science at the Regional Engineering College of the University of Allahabad, India.

Shaowen Su is a graduate student at the Department of Computer Sciences at the University of Texas at Austin. He received his MS in Engineering Physics from the University of Oklahoma and his BS in Physics from Beijing University, Beijing, P.R.China.

Simon S. Lam is chairman of the Department of Computer Sciences, University of Texas at Austin, and holds two endowed professorships. Prior to joining the University of Texas at Austin faculty in 1977, he was a research staff member at the IBM T.J. Watson Research Center, Yorktown Heights, New York from 1974 to 1977. His research interests are in the areas of computer networks, communication protocols, performance evaluation, formal verification, and network security.

Simon received the BSEE degree with Distinction from Washington State University in 1969, and the MS and Ph.D. degrees in engineering from the University of California at Los Angeles in 1970 and 1974, respectively. He was a recipient of the 1975 Leonard G. Abraham Prize Paper Award from the IEEE Communications Society. He organized and was program chair of the first ACM SIGCOMM Symposium held at the University of Texas at Austin in 1983. He presently serves on the editorial boards of *IEEE Transactions on Software Engineering* and *IEEE/ACM Transactions on Networking*.