PAVING THE ROAD TO NETWORK SECURITY OR THE VALUE OF SMALL COBBLESTONES

Hilarie Orman
Sean O'Malley
Richard Schroeppel
David Schwartz [1]

TR 94 16

Abstract

Software subsystems that implement cryptographic security features can be built from small modules using uniform interfaces. The methods demonstrated in this paper illustrate how configuration flexibility can be achieved and how complex services can be constructed, all using the same building block modules. These allow the configuration process to be independent of algorithm details, while the algorithms used in the subsystem are obvious.

May 23, 1994

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the National Computer Security Center Grant MDA904-92-C-5151. This paper appears in ISOC'94. Authors' email addresses are ho, sean, rcs, dcs @cs.arizona.edu

1 INTRODUCTION

Adding security to Internet protocols is a most worthy goal, one that is simultaneously easy and hard. It is often easy because there are many good ideas and existing approaches; it is hard because it is difficult to modify existing protocols or add new ones across a variety of platforms. We advocate an approach that does not dictate a solution in terms of mechanisms, but instead dictates a software architecture that is malleable, extensible, and shaped towards the concerns of designers of high assurance systems for privacy and authentication.

We have been experimenting with highly structured protocol implementations for security-related protocols. The vehicle for the research is an object-oriented architecture for building communication systems: the *x*-kernel [8]. We have been diligent in our efforts to add modular security enhancements, using the *x*-kernel interface. Our goals have been to discover the architectural changes needed to support security-related functions, and to determine the impact of implementing privacy and authentication functions as protocol objects in an object-oriented framework.
There are some interesting preliminary results. We have investigated three areas:

- optionless protocol layers for applying cryptographic enhancements
- datagram security at the network and transport levels
- implementing Kerberos in modules

These investigations have shown that while there are some problems with the most common Internet protocol suites that interfere with modularity and layering, a wide variety of enhanced network configurations are easily available when the cryptographic support library is added. A result of our work is that although a standard protocol interface suffices for our implementations, some novel combining modules are necessary to accommodate the complex message types. The design of messages for authentication protocols, in fact, has a substantial effect on the structure of the software that processes them. Overall, our object-oriented approach to protocols makes adding cryptographic objects to standard communication suites easy.

2 Uniform implementation framework

An architecture for building flexible communications systems can go a long way towards promulgating good network security standards. Today's reality, though, is that the monolithic communications systems built into operating systems are the cause of the ossification of the networking utilities. They retard deployment of new protocols for security enhancements for these reasons:

- The base protocol set cannot be extended without kernel modifications that are frequently difficult to execute and require access to proprietary code.
- There is no single point of control for a communication configuration for a host or a group of hosts. Many systems have conglomeration of services such as vendor-supplied network security services, Kerberos [11] added to some programs, auditing options, protocol daemon wrappers, and Privacy Enhanced Mail with associated certificate hierarchies [12, 10, 4, 9].
- Changing a network configuration can cause havoc because of the difficulty of describing the change and its impact on individual host configurations.

We have been developing the infrastructure for network authentication and access control using objects with uniform interfaces. This has proven effective for building efficient transport services, and it appears to be equally effective and convenient for layered privacy and authentication services. The uniform interface means that modular services can be inserted easily into a protocol graph to provide security enhancements at almost any level. The configuration is based on a single, succinct text file that describes the protocol graph, including security protocols.

We have been working with the Mach kernel and Unix server code to incorporate the architecture into the standard user environment. In addition, the code is written to allow its use as part of the Mach kernel or as a network server. The latter approach allows the system administrator to easily change the communication server configuration.

3 LITTLE OBJECT-ORIENTED PROTOCOLS

The objects used for building a communication service are protocols, and they support functions initiating sessions, sending and delivering messages, and terminating sessions. Unlike more traditional methods of protocols design that have a high ratio of code to state (as does TCP), our protocol objects typically have a simple, well-defined network function implemented in a few pages of C code, and their interface to other protocols is generic enough that they can be composed with others to build complex functionality.

For privacy and integrity enhancements, we build privacy out of a stack of modules, each of which performs one function. Each module has the uniform protocol interface which characterizes all our communication functions.

For authentication functions, such as provided by Kerberos, the complex functionality is constructed from small building blocks organized in a graph with many branches. The result deliberately looks something like a dataflow diagram for designing Kerberos.

3.1 Optionless cryptographic protocols

The first part of our work has been to develop the basic building blocks of secure communication, i.e. the cryptographic algorithms. Although it is traditional to view these as subroutines that operate on message data buffers, we have chosen to represent them as protocols, for several reasons. First, messages flow along a protocol graph that is static; this means that we can gain assurance that data is being encrypted before transmission by analyzing the graph and looking for links that might bypass the encryption modules. Second, protocols are highly identifiable objects in our system, and they have limited means of interaction. This allows us to analyze most information flow issues locally. Third, by enforcing the strong separation of message interpretation from message encryption, we gain a measure of confidence that we can substitute alternative encryption schemes with little or no redesign of the control structure.

We have used the *x*-kernel architecture for this work. The *x*-kernel view of a protocol is as an object with a small set of interface functions that are tailored to its specific purpose (e.g. local network datagrams, remote procedure call, internetwork datagrams, windowing mechanisms, etc.). A protocol keeps its connection state in objects called *sessions*; a protocol demultiplexes incoming messages to sessions, and messages are sent to remote destinations by being "pushed" to sessions along a path in the protocol graph. Addresses are also objects; the representation of an address is chosen to match the protocol that will open the connection.

Protocols are software modules that organized into a directed, acylic graph structure for purposes of applying interface functions. The uniform interface rules allow a protocol to send a message to a protocol below it in the graph, or to demultiplex a message to a protocol higher in the graph. The operations occur within the context of an execution unit that is a thread. Threads, messages, protocols, addresses and sessions are all first-class objects in the *x*-kernel.

The building block protocols for our security work are cryptographic protocols with minimal option sets. Each protocol performs a specific algorithm. The header information is limited to that which is necessary to the correct functioning of the algorithm, and some algorithms require no header. Our goal was to develop building blocks that were efficient and easily inserted into almost any part of the protocol graph. Each cryptographic protocol contains only a minimal structure for maintaining network state information; a substantial amount of code is, of course, devoted to the complex algorithms for computing data encryption or one-way hash functions.

We have implemented seven cryptographic protocols:

confounder This puts 4 random bytes at the head of a message to thwart known plaintext attacks. This is similar to "salt".

keyed confounder This chooses an initial value to be used with a one-way hash function. The protocol directs its lower protocol to use the initial value for all messages on the associated network association.

DES in CBC mode Two distinct variants are provided: length preserving and padding to eight byte boundaries [5, 13].

MD5 One-way hash function from RSADSI.

SHA NIST one-way hash function [2].

DSS The NIST-proposed Digital Signature Standard [1].

RSA Public Key The exponentiation scheme using public keys [16].

Diffie-Hellman Key Exchange A public key method for negotiating private keys[6]. This will be augmented with protocols supporting signed message exchanges [7].

These protocols have uniform interface requirements and can be composed in the protocol graph with great freedom. In particular, the stack of confounder, hash code, and encryption provides messages that are private and resist modification. The "open" operation on a protocol supplies a generalized destination address that is used to select an appropriate key, if required. Examples of "addresses" include 32 bit internet address, a user identification string, or a (UDP port,IP address) pair. A simple protocol like confounder uses no address information, of course.

A simple use of these functions for applying privacy and integrity at the network level is illustrated in figure 1. The routing and network packet functions (e.g., ARP and IP) process messages that would normally come from or go to the network interface. Cryptographic protocols can be interposed without modifying the pre-existing protocols; the protocols add information or encrypt data on its way to the network, and they remove information or decrypt when messages are received from the network. In the example shown,

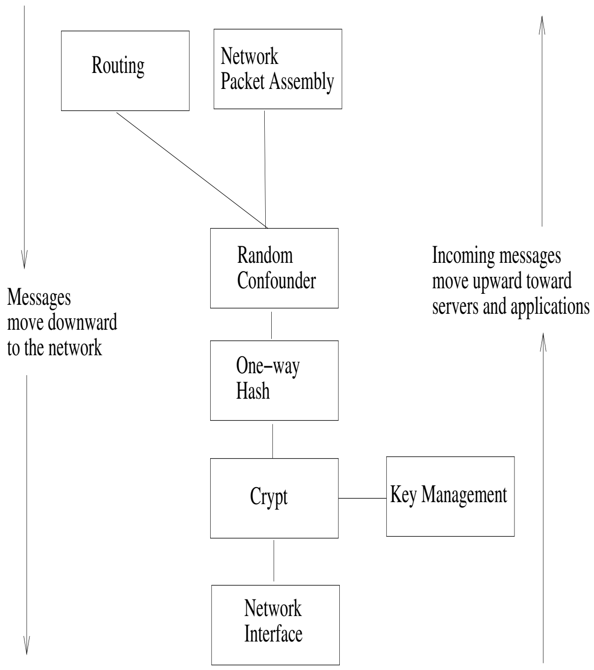

Figure 1: Simple composition of cryptographic protocols

a confounder protocol puts a random byte string in the message, the one-hash function puts a hash value into the message, and the "crypt" function, using the destination address and the key manager, encrypts the data. In the reverse direction, the message is decrypted using a key associated with the source address, the hash value is validated, and the random byte string is removed. Either MD5 or SHA can be selected for instantiating the hash function, and either DES or RSA can be selected for encryption. These instantiations are selected at system build time; runtime algorithm selection can be accomplished with additional protocol modules.

3.2 Restrictions on simple composition

The only modularity problem encountered at this level concerned the blocking factor of 8-bytes per DES encryption step. A straightforward DES implementation alters the data length by padding it to a multiple of 8 bytes. For network purposes, the true data length must be included as auxiliary information, usually a header. One of the DES protocols maintains the exact length of the data using "ciphertext-stealing" for short blocks [13]; this avoids some of the problems with the IP pseudo-header (see next section). This variant must be presented with at least 8 bytes of data, but otherwise has no restrictions. The result of the bifurcation of DES methods is that subtle correctness conditions are introduced that impose new limits on using the DES protocol building block.

Another restriction on composition is that the protocols can only be used as services underlying datagram protocols. This means that all of the message must be present in order for the recipient to invert the enhancements, and the protocol state is not modified by processing a message. Streaming protocols such as TCP require a different model of operation that either inserts markers for message boundaries or keeps state information. The first method admits a layered solution, but the second is a more natural match with stream semantics.

Using optionless protocols adds the requirement that all hosts in the domain of protection must have exactly the same protocol configuration. There are two reasons for accepting the limitation: our configuration method is so simple that system administrators could easily bring all hosts in a domain into coherence, and because the negotiation or interpretation of options could be delegated to a separate protocol layer that provides an umbrella over the option suite.

3.3 Composing optionless protocols with other Internet protocols

When the padding version of DES is placed in the graph between IP and either TCP or UDP, the pseudo-header daemon rears its head. The IP pseudo-header is a weak integrity mechanism for IP clients. The clients compute a checksum over the IP header and client data. Sending clients build an IP header that they expect IP to use, and receiving clients use the received IP header, and the integrity check is performed against the checksum which was included in the received message. If an intermediate protocol changes the data length, then the received IP header length field will not match the actual data length, and the checksum will fail. This problem is a ramification of the long-observed and much lamented lack of clean layering between IP and its most common clients.

When cryptographic enhancements that modify message length are interposed over IP, they must adjust the length field of the received IP header to match the length of the received message minus the padding or other added information. This adjustment occurs after IP processing, but before delivery to TCP or UDP. In order to alert the protocols to this additional role, which is dependent on its graph position, we use a "control" operation. In the *x*-kernel, control operations are propagated down the protocol graph until a protocol recognizes the operation code and returns a value. TCP and UDP use this operation to alert each of the lower-level cryptographic protocols to the fact it is situated over IP and must perform the length correction. Thereafter, after successful decryption, the encryption protocol adjusts the length field in the copy of the IP header that was attached to the data.

The protocols must be careful to preserve the correctness of auxiliary network specific information. In figure 1, if the encryption protocol does not maintain the exact length of the data, the upper protocols may choose packet sizes suboptimally because they are unaware of the overhead added by the encryption protocol. The *x*-kernel protocols determine the optimal packet size by making inquiries of lower protocols, and this is done recursively. Therefore, the cryptographic protocols subtract their overhead requirements from the packet size before returning the recommended packet size.

3.4 Key management

Cryptographic protocols requiring keys have a graph link to a key manager "protocol," which is really a database manager. Its purpose is to hide the abstraction of association keys from the encryption protocols; the protocols supply the data structure with the association key (internet address, ethernet address, etc.) and the key manager produces a key. The secure protocol for importing, changing or revoking keys must have a link to this manager.

The key manager provides the association between the network addressing used by the cryptographic protocol and the keys that control the algorithm. When used in conjunction with protocols for ethernet packet encryption, the addresses are 6-byte ethernet addresses; when used for IP security, the addresses are

32-bit IP addresses. As explained later, Kerberos uses a two-level scheme to achieve the binding of a user identifier to a session key.

The key management layer is intended for use with a protocol for key updates, revocations, additions, etc. The evolving set of interface operations for the key manager supports these operations. The policies regarding keys and the mechanisms for obtaining them will reside in the keying protocols.

4 NETWORK AND TRANSPORT LEVEL SECURITY

4.1 Modes of operation

The use of cryptographic protocols in the *x*-kernel framework is illustrated in this section with four configurations, depicted in the *x*-kernel graph notation. In each case, a graph that implements a standard Internet protocol configuration is shown with the addition of cryptographic enhancement modules.

The *x*-kernel protocol graph diagrams used in this paper show the possible paths that can be taken by a message as it moves from the application services (at the top of the graph) to the network (at the bottom of the graph) or as an incoming network message is moved through successive processing and demultiplexing stages. The graph nodes, represented by rectangles or rotated squares (or "diamonds"), are the protocols, e.g. TCP, IP, etc. The *x*-kernel diagrams use diamond-shaped nodes to represent "virtual protocols", i.e. protocols that act as switches for selecting the path a message will take; the switching decision is based on the address, the message size, or other attributes that are germane to the local environment but outside the purview of the communication protocols themselves. For example, the internet packet protocol (IP) is implemented as two protocols: one implements the usual fragmentation and reassembly functions, and the second, virtual protocol (VNET) sends the packet to the correct network interface, based on the destination address.

Four illustrative configurations for providing network level security have been investigated. These illustrate how the building block approach provides the flexibility to configure network security solutions:

- A security level at the network interface. By interposing a security protocol between the network driver and anything above it, the enhancements are applied to all packets leaving the machine along that interface. This can provide, for example, privacy along ethernet segments that are otherwise vulnerable to tapping. A recommended configuration would use a confounder and encryption. This is illustrated in the lower part of figure 3.

- Applying security enhancements to non-local traffic. By interposing a security module above the internet packet level (IP) level, one can assure that packets with destinations that are not on a locally connected network are enhanced. This would be appropriate if the locally attached network hosts (including gateways) were trusted. This is illustrated as part of figure 3, by adding cryptographic enhancements above the IP protocol.

- Encapsulation of network packets. By encapsulating non-local, encrypted traffic inside another layer of network addressing, we can provide an obfuscation of the visible portions of the network header. These packets may require decryption at cooperating gateways in order to correctly route them, but they are opaque from point to point. In this case, each host is responsible for applying the crypto enhancements. See figure 4.

- The encapsulating gateway. An alternative to the previous graph could be used on a gateway machine to treat one attached network as secure, and another as vulnerable. The protocol graph would apply the crypto enhancements to only the packets destined for the vulnerable network. The gateway is then the only encrypting host on a local network.

To illustrate the ways that building blocks can be configured, the next sections show how the cryptographic enhancements can be added to a standard local area network protocol graph, one that has optimizations that bypass the IP layer for messages that are destined for local delivery. The BLAST protocol [15] is used for fragmenting large local messages (those that exceed the maximum transmission unit for the network interface).

4.2 The security selection virtual protocol

Not all network security configurations are as static as the ones outlined above. Where a system is configured to participate in more than one style of security, a multiplexing protocol must be used to select the style that matches the destination; this amounts to setting up a data flow through the subgraph responsible for the appropriate enhancements. A trustworthy protocol could be used to selectively apply security enhancements to messages based on the destination address. This would allow, for example, a system administrator to apply particular enhancements to messages addressed to a UDP service port on a remote machine, and/or to expect security enhancements on messages directed at incoming service requests on UDP service ports. Some system administrators have suggested that news or bulletin board services such as NNTP could be treated this way to provide a degree of authentication between mutually trusted sites.

The Security Selection Virtual Protocol is a concept that is evolving as part of our research. Its placement in a communications graph is illustrated in figures 2 and 5. The diagrams used dashed lines to indicate modules that have security crucial code.

The design of this protocol is deliberately simple and involves no negotiation or setup of security contexts. A protocol such as NLSP 4.3 can be used for situations where more dynamic contexts must be supported.

4.3 Standards: NLSP

The examples so far have shown how simple protocols can be used for network security, but the simple protocols are not standards adopted by an organization. The simple protocols are, however, appropriate for constructing protocols that are defined by external standards. Our building blocks concept can support the implementation of cleanly layered protocols such as NLSP [3]. The sample configurations from the "Modes of Operation" section are in the spirit of such a protocol. They provide privacy and integrity via composition of DES and MD5. Authentication of the source address is a side effect of the integrity options, and encapsulation can be provided as illustrated in the preceding section.

NLSP is, however, more complicated than a static layering of message enhancements. The negotiation of security mechanisms and the protocol for setting up security association contexts are things that make NLSP a comparatively "big" protocol. An implementation in the *x*-kernel architecture would involve decomposition into a graph of smaller protocols: the cryptographic protocols, NLSP processing for each message type, a security context manager, and the Security Association Protocol (both inband and out-of-band). The Security Association Protocol itself would require a separate subgraph of supporting cryptographic protocols.

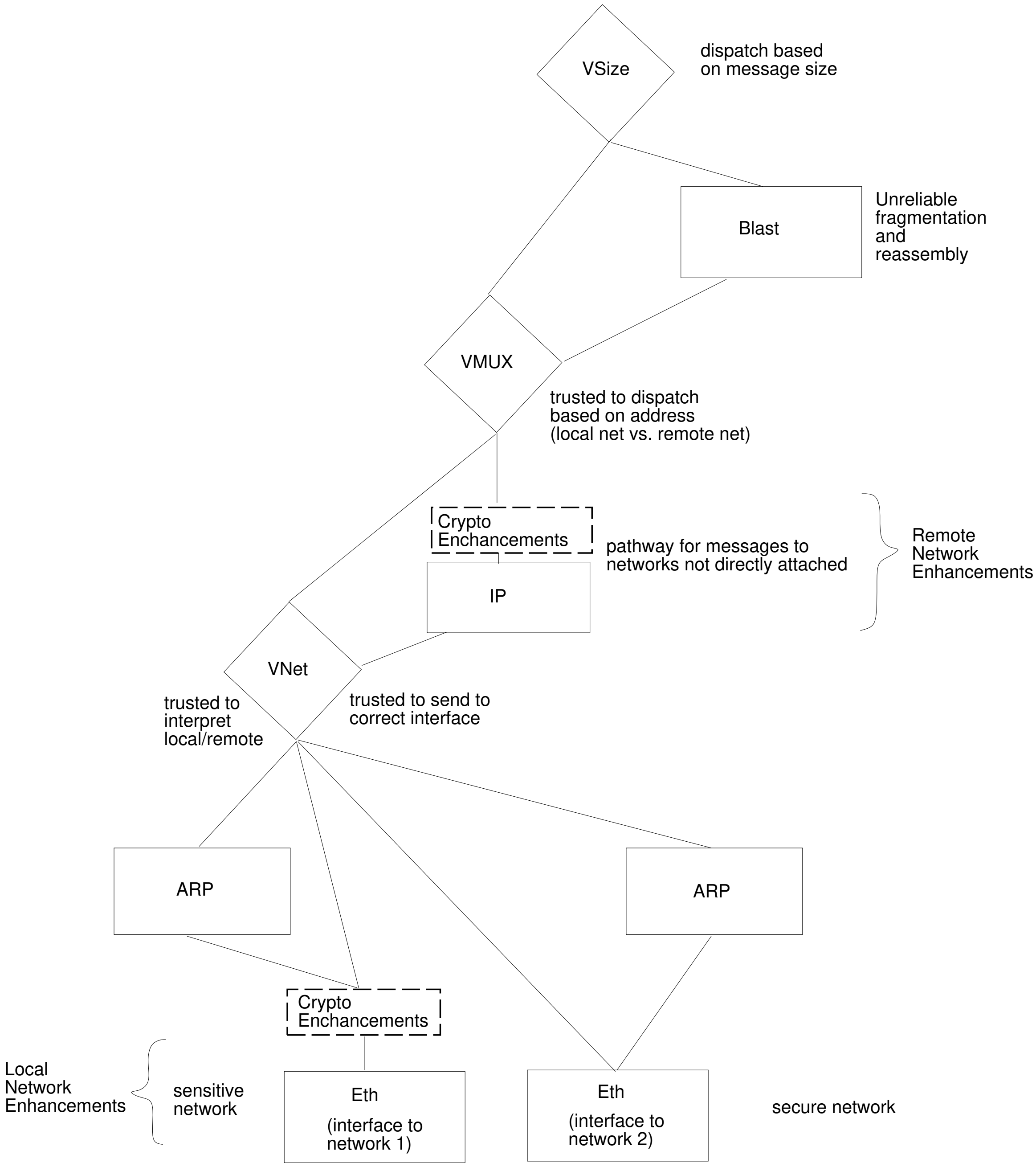

Figure 3: Two options for adding network level security

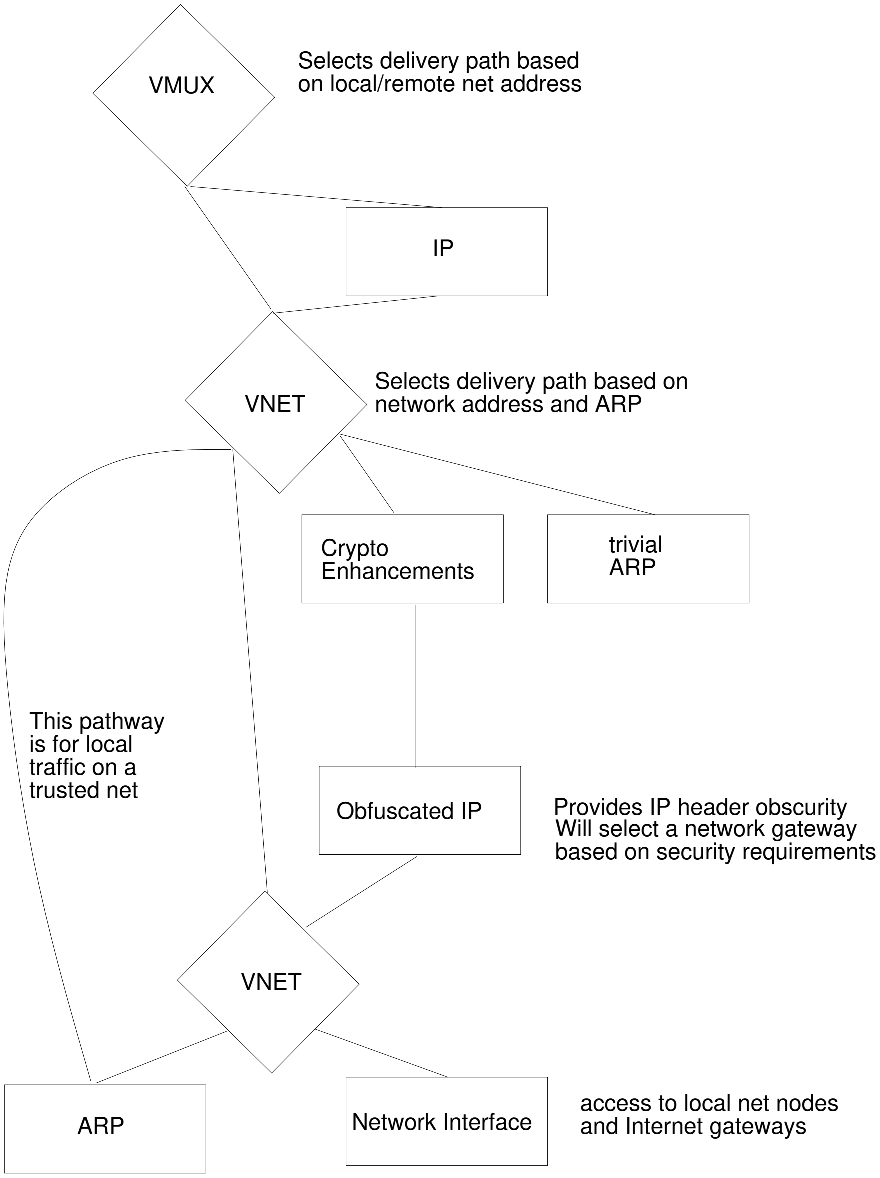

Figure 4: Internet Packet Encapsulation

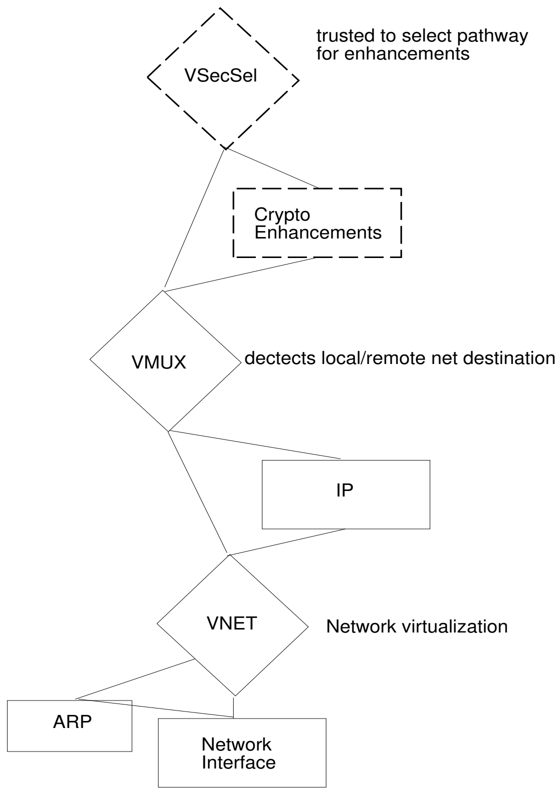

Figure 2: Using an address-based scheme to select security enhancements

4.4 Securing a protocol graph

We have developed protocol graphs that secure IP or ethernet communication, using one key for each host that we are communicating with. Does this create a secure system? If one can draw a horizontal line across the graph and see that everything above the line is a trustworthy encryption protocol, then the system as a whole has some pretense to being trustworthy. The trust is dependent on the trustworthiness of the encryption software and the *x*-kernel infrastructure.

In other cases only some of the traffic might be encrypted, and the the protocol graph would be configured to provided security for a class of traffic. For example, in special circumstances, one might secure IP but not the ethernet, on the assumption that the local cable is trusted (and all gateways are trusted). In other circumstances, one might have a system that secures some services and not others. For example, TCP but not UDP, or some TCP ports might be secured and not others. In these cases, the modules that choose the network interface or interpret the TCP and UDP headers must be trustworthy. The protocol graph in this case is useful in identifying places where trust is placed.

For mixed mode communication, there can be filter modules that are trusted to distinguish trusted from untrusted traffic. For example, a virtual network layer might have a list of hosts that are permitted to have non-secured communication, or it would have a list of hosts that must have secure communication because the local site sends sensitive data to them. The trustworthiness of the filter module is the major issue for these systems, and the protocol graph itself can suggest how the filter module may be dependent on addressing or

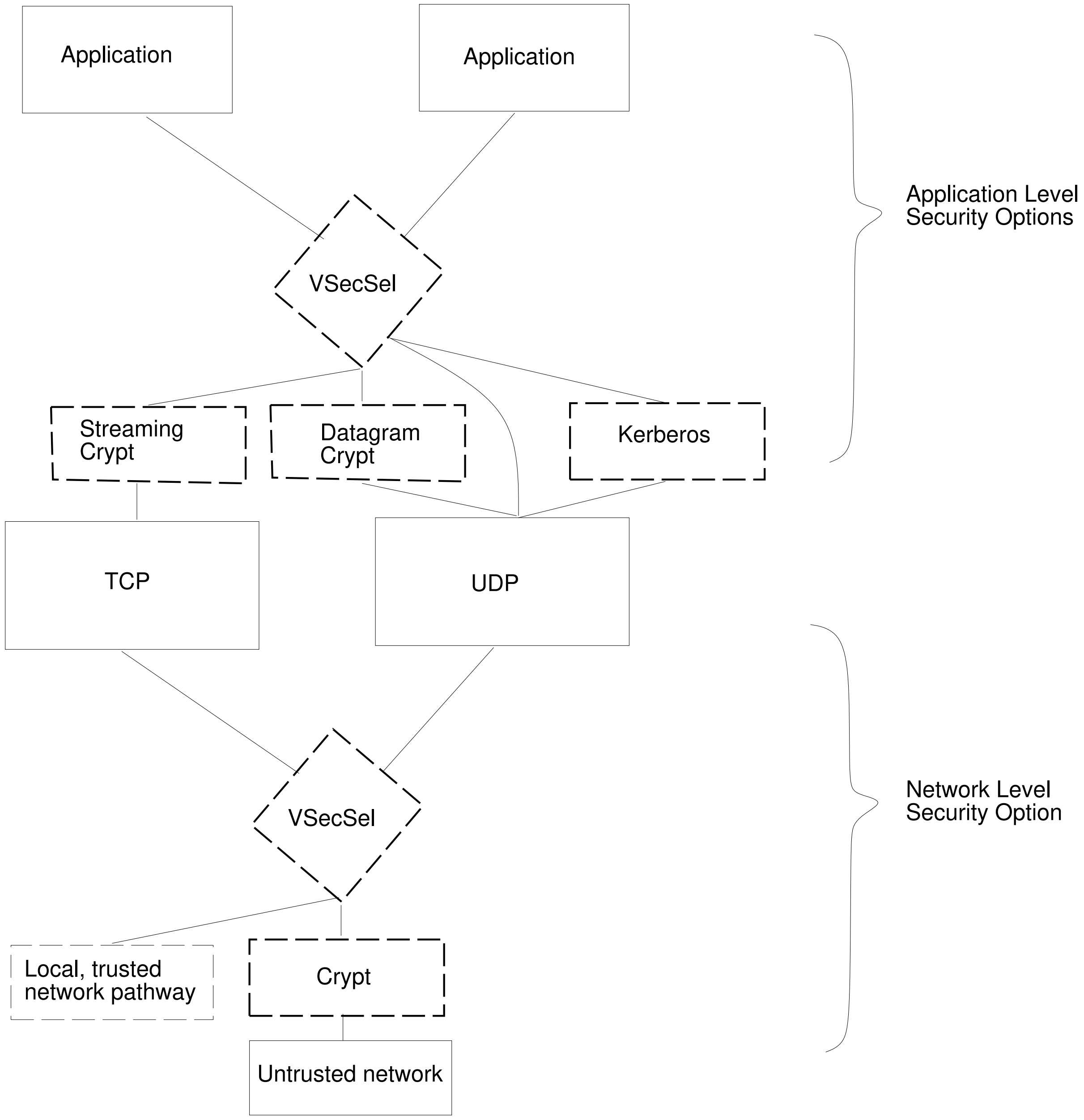

Figure 5: The security enhancements based on application type

other services of lower protocols.

5 DECOMPOSING KERBEROS

If the building block design is done well, it should be useful in building complex applications, such as Kerberos. In developing a decomposition of Kerberos 5, we have modeled Kerberos as a client/server protocol supported by cryptographic building blocks. Kerberos is not strictly a protocol — it is more like a collection of message types, but we have persevered in our viewpoint. The cryptographic dependencies of Kerberos 5 are largely hidden in the MIT implementation, and only a careful reading of the source code reveals exactly which enhancements are applied to which data. Our goal in implementing Kerberos 5 with our cryptographic building blocks has been to thoroughly expose the data flow and application of enhancements.

We have developed a completely modular version of Kerberos 5 in order to validate our concept. All encryption and key management is handled by instances of cryptographic and key management modules. Because a single Kerberos message is composed of cleartext and encrypted portions, we use a protocol for splitting and joining the message pieces to direct them through the protocol graph. This data flow is contrary to traditional layered protocol design, because a single message must be processed in separate but dependent parts.

The heart of the Needham-Schroeder authentication method [14] and its many derivatives dictates that an authenticatable message must be processed in two passes: one that uses a pre-existing key and one that uses a key encrypted in the pre-existing key. As a result, the message processing must first split the message, then en/decrypt one part, send the key to the graph section responsible for en/decrypting the next part, and finally en/decrypt the next part.

A similar additional complication arises in doing checksum validation for Kerberos 5 messages, because the checksum of a cleartext message part is encrypted with a pre-existing key. The checksum must be decrypted before the cleartext can be validated.

We have used the MIT Kerberos 5 subroutines for building the cleartext of messages for authentication requests and ticket requests. However, we do not use the Kerberos encryption library. Instead, we send the Kerberos message out from the authentication protocol in three pieces. The two pieces requiring cryptographic enhancement travel through the protocol graph to the cryptographic protocol modules, and the cleartext piece moves directly to the assembly protocol layer, which joins the pieces together. When the authentication reply message arrives at its destination, the three parts are separated. Of the three parts of the reply message, only one is decrypted by the client. One part contains a cleartext message from the authentication server, and another part contains encrypted data that can be retained and forwarded, but not read by the client (this last part is the Kerberos "ticket.")

The result of the decomposition is shown in figure 6, which depicts the graph used by a Kerberos server protocol for issuing tickets. The arrows indicate the processing of an incoming request; the reply is processed by the same graph. This graph is extraordinarily ugly; some of the complexity derives from the two-level decryption implied by using tickets and encrypted authenticators, but much of the problem arises from the Kerberos message structure itself.

First, we will examine the relatively clean subgraph section, then the complicating factors.

5.1 Kerberos message protection

Kerberos uses three techniques to protect encrypted data: a confounder, a checksum, and DES encryption. We apply these enhancements by passing the message through three protocol layers, each one devoted to the appropriate algorithm. The layers are not aware of the Kerberos message structure; they treat the data as an untyped byte stream. The diagram illustrates this protocol stack as a tightly bound triple: confounder, MD5, DES.

5.2 Kerberos key and ticket management

The Kerberos domain administrator establishes a set of keys that are associated with user or service names. But in operation, these identifiers are associated with network connections, and these are named by network addresses. This leads to a two-level binding scheme, and we have used two instances of the key manager to accomplish this. Thus, a request from a client "joeluser" at IP address 192.12.69.222, port 7001, causes the authentication server to first find the key for "joeluser" and then bind the transport connection (192.12.69.222, 7001) to that key. The first association (user to key) is relatively long-lived, the second is transitory, lasting only as long as it takes to get the reply back to the user.

Tickets are opaque objects to clients. They are referenced by service name and encrypted in a key shared between the ticket granting service and the server. The servers require that a ticket be presented by the client during an initial "service open" request. The server uses the ticket to form an association between the server name and the source of the client request. This allows decryption of the ticket. Our protocol graph for Kerberos servers includes a stack of protocols that reverse the cryptographic enhancements for the ticket (decryption, checksum validation, removal of confounder). The Kerberos clients do not have these modules in their protocol graph.

5.3 Data encoding vs. modularity

The Kerberos server diagram has two areas that indicate how the Kerberos data presentation interferes with our attempts to decompose the protocol. They are both indicated by occurrence of data encoding modules, once as part of the split/join/code layer, and once in the area of of the "copy" and "encode" sequence. In both these areas, incoming data must be moved out of its network representation as a linear byte stream and into its machine dependent data structure representation.

Kerberos uses ASN.1 for the network encoding, and for the most part this data presentation method works well with the *x*-kernel. However, our message splitting and joining module should be largely independent of the message encoding semantics. The module needs to be able to splice together linear pieces of previously encoded data into a single, encoded message, and to reverse this structuring at the receiving node. To our knowledge, ASN.1 does not support this sort of splicing, i.e.:

$$encode(encode(A), encode(B)) \neq encode(A, B)$$

Instead, in order to have our Kerberos implementation interoperate with others, our split/join module must have knowledge of the ASN.1 encoding details in order to implement:

$$splice(encode(A), encode(B)) = encode(A, B)$$

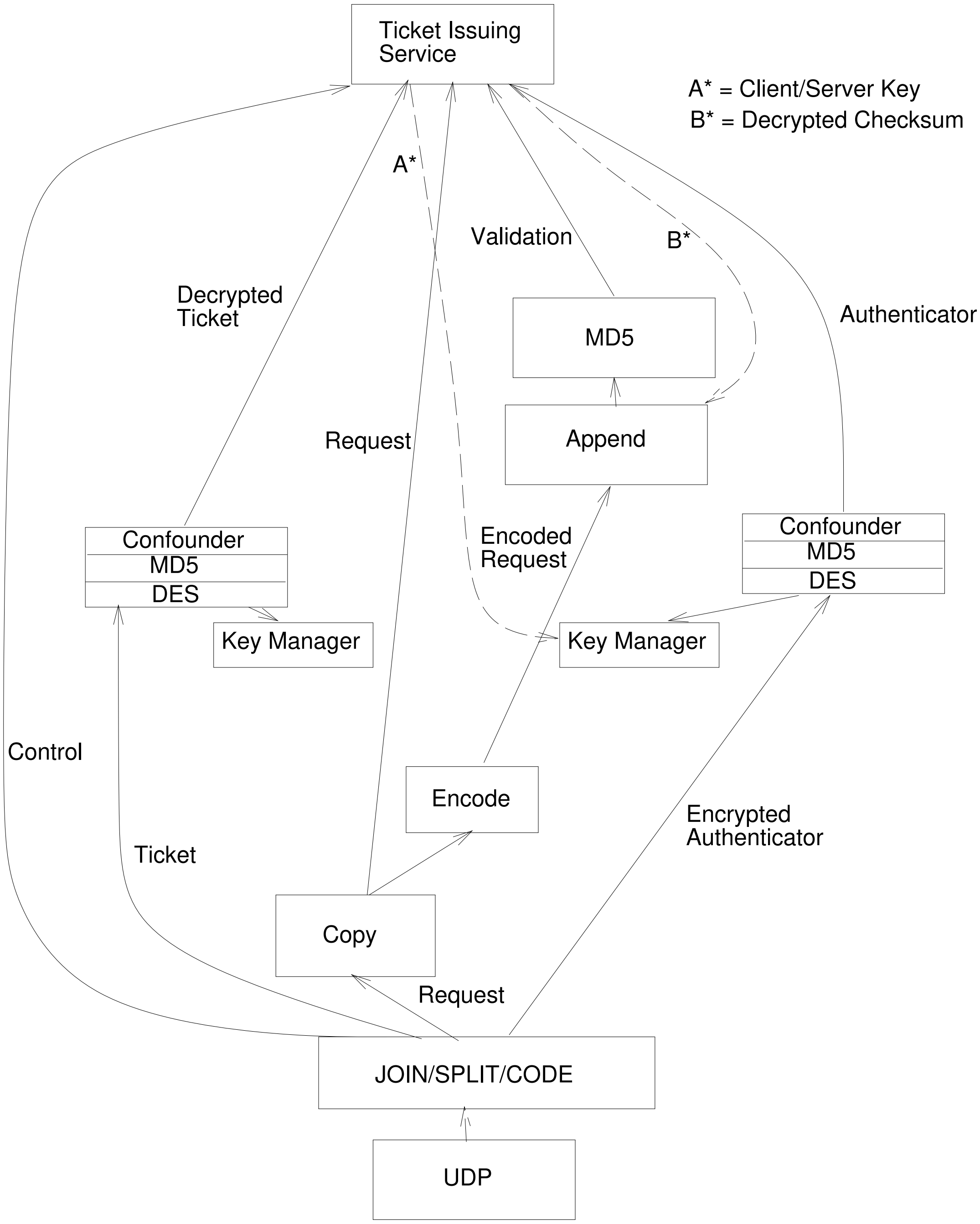

Figure 6: Kerberos ticket request: server side processing of incoming client request

Using the notation $encrypt(A, K)$ to mean datum A encrypted with key K, a typical Kerberos message can be viewed as:

encode(cleartext;
    encrypt(encode(A); K1);
    encrypt(encode(K1); K2)
)

There are two problems with this: the last part of the message must be processed first, and the encrypted portions are not readily accessible as message pieces. Instead, the entire message must be decoded into constituent parts before the encrypted portions can be identified. A more natural way of dealing with encrypted data would be:

encodesimple(
        encode(cleartext);
        encrypt(encode(K2); K1);
        encrypt(encode(A); K2)
)

In this representation, the *encodesimple* layer need know nothing about the detailed structure of the cleartext, it need only know the length of the encrypted data. Systems like the *x*-kernel always know the size of their data areas, and they can use this as input to the *encodesimple* routine. In addition, the message can be processed in the natural order of computation, left to right.

One might wonder why we spend so much time examining the details of message representation. The reason is that it affects the software engineering and the efficiency. It is easier to engineer good software over good data representations, and that leads to higher assurance that the software is correct. The efficiency of data processing is a major factor in heavily utilized, high speed networks. We seek to design systems that can handle a large number of requests and a large amount of data with great aplomb.

5.4 Compatibility issues

We expect to be able to interoperate with other Kerberos implementations, but we raise two issues about the MIT implementation. One concerns the location of the confounder in the representation of encrypted data. If the confounder were placed after the message checksum instead of before it, it would be easier to have the confounder be a processing layer in the protocol graph, and it would be easier to calculate the checksum. The logic requiring the confounder at the beginning of the message is predicated on the requirement that unpredictable data occur first, to thwart known plaintext attacks. However, if a strong checksum algorithm is applied to data containing the confounder, the checksum should be equally unpredictable and thus suitable as the starting value for the encrypted portion.

The second issue concerns the informal protocol used by Kerberos to send messages via UDP. It first sends a short descriptor giving the message type and length, and then sends the entire message. This exchange seems *ad hoc* and directed at particular Unix socket implementations. We can implement a protocol layer that implements this exchange, but we would feel more comfortable with it if it were more formally defined as being part of Kerberos (rather than just something we observed in the MIT source code).

Other client/server Kerberos interactions are *ad hoc* conventions between the two parties. A formal protocol definition that delineated exactly how the initial request/reply affected subsequent network traffic would be a big step towards generic network security.

6 CONCLUSIONS

There are two basic conclusions: the *x*-kernel is a useful vehicle for quick prototyping of secure communication systems, and decomposition into small modules is a powerful way to represent these systems, especially for the network layer.

We have been able to implement a number of security-relevant architectures using the *x*-kernel, and this has been done without any modifications to the *x*-kernel infrastructure. Some usage rules have been developed, and new protocol objects have been developed, and these can be configured for use and reuse in communication suites.

Kerberos seems amenable to being broken into a small set of cooperating protocol modules, and this approach seems relatively generic. However, the details of message structure and network encoding have a dramatic effect on the complexity of the resulting decomposition. A clear separation between message composition and data structure encoding is necessary for a truly modular, complex security service.

REFERENCES

References

[1] DRAFT specification for a digital signature standard (DSS). Technical report, Federal Information Processing Publication, August 1991.

[2] DRAFT specification for a secure hash standard (SHS). Technical report, Federal Information Processing Publication, January 1992.

[3] ISO-IEC DIS 11577 - information technology - telecommunications and information exchange between systems - network layer security protocol. Technical report, ISO/IEC JTC1/SC6, November 1992.

[4] D. Balenson. Privacy enhancement for Internet electronic mail: part III algorithms modes, and identifiers. RFC 1423, February 1993.

[5] American national standard data encryption standard. Technical Report ANSI X3.92-1981, American National Standards Institute (ANSI), December 1980.

[6] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information*, (IT-22):644–654, November 1976.

[7] W. Diffie, P. C. V. Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes, and Cryptography*, 2:107–125, 1992.

[8] N. Hutchinson, L. Peterson, S. O'Malley, E. Menze, and H. Orman. *The x-Kernel Programmer's Manual (version 3.2)*. Computer Science Department, University of Arizona, Tucson, Arizona, January 1992.

[9] B. Kaliski. Privacy enhancement for Internet electronic mail: part IV key certification and related services. RFC 1424, February 1993.

[10] S. Kent. Privacy enhancement for Internet electronic mail: part II – certificate-based key management. RFC 1422, February 1993.

[11] J. Kohl and B. C. Neuman. The Kerberos network authentication service (V5). Technical report, Internet Engineering Task Force (IETF), FTP information: rs.internic.net /rfc/rfc1510.txt, April 1993.

[12] J. Linn. Privacy enhancement for Internet electronic mail: part I – message encipherment and authentication procedures. RFC 1421, February 1993.

[13] C. H. Meyer and S. M. Matyas. *Cryptography: A New Dimension in Computer Data Security*. John Wiley and Sons, 1982.

[14] R. M. Needham. Using cryptography for authentication. pages 103–132. in the book Distributed Systems, published by ACM Press, New York, New York, 1989.

[15] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Trans. Comput. Syst.*, 10(2):110–143, May 1992.

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