Evolving an Implementation of a Network Level Security Protocol[1]

Hilarie Orman
University of Arizona[2]

TR 95 15

Abstract

While the importance of developing software for privacy and authentication on the Internet can hardly be gainsaid today, the process of bringing standards to fruition can be a long and arduous one. We have used prototyping methods to track the efforts of the Internet Engineering Task Force network security group (IPSEC), concentrating on the software challenges and advantages posed by the standards. As a result, we have some observations and useful guidelines for current and future developers of secure networking software.

January 24, 1996

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

[1] This work was supported by ARPA under the grant DABT63-91-C-0030.
[2] Author's address: Department of Computer Science, University of Arizona, Tucson, AZ 85721. Email: ho@cs.arizona.edu.

1 Forward

While the importance of developing software for privacy and authentication on the Internet can hardly be gainsaid today, the process of bringing standards to fruition can be a long and arduous one. We have used prototyping methods to track the efforts of the Internet Engineering Task Force network security group (IPSEC), concentrating on the software challenges and advantages posed by the standards. As a result, we have some observations and useful guidelines for current and future developers of secure networking software.

One of the successful results of using our architectural infrastructure for network protocols was that, for a prototyping project, we had to discard remarkably little code. We did, however, experience "exponential option-ism" and control structure overload. This had led to our maxims for prototyping and evolving secure network software:

- Surround the cryptographic algorithms with an overly general interface to the message processing routines.
- Keep module interfaces "typeless" in terms of addressable network entities.
- Use a protocol control structure that is more flexible than the final system will require.

This paper demonstrates how an increasingly sophisticated set of prototypes culminated in software implementing a standard and how the experience supports these recommendations.

2 Introduction

Our project members have developed a continuous stream of implementations of network level security prototypes during the last three years, in parallel with standards development of the Internet Engineering Task Force IPSEC working group. The proposed standards which have finally emerged as RFC's 1825-1827 [1], with several related RFC's defining related cryptographic processing [5] (we will refer to the collection of RFC's as the IPSEC protocol). In addition, we have tracked evolution of a key exchange protocol, Photuris, currently an Internet draft [10].

The Internet development effort coincided with our own research into using highly modular software for network security services, and one of our tracks of effort centered on the IETF definition of network layer security. Our prototyping work began with simple mechanisms, and these have been the foundation software which has converged with the standard, up to the point of beginning to demonstrate interoperability with other implementors.

The goal of the network security protocol is to protect datagrams traveling over the Internet. The protection includes authentication, integrity, and privacy. For our purposes, having a basis for authentication between IP addresses is sufficient. The protocol is independent of the basis of authentication, that responsibility being delegated to the key establishment protocol, which can be either some form of out-of-band secure distribution or a protocol such as Photuris [10].

2.1 Our Goals

The question we wanted to examine was whether there was a a uniform interface between protocol modules that was sufficient and efficient for building complex network security systems. We had three goals:

- a simple interface
- a fine-grained definition of modularity
- an analyzable system

Overall, we have maintained adherence to these principles during the period of development. Nonetheless, the interconnection structure of the modules changed in some unexpected ways. We recommend these structural extensions to others who may come after us and aspire to network code that is both efficient and constructed for high assurance analysis.

- Highly flexible control structure for message processing
- Nearly typeless modules
- Generalized storage management

Network security software is made of many small pieces comprising a large system; in our earlier work on implementing Kerberos in small modules [9], we worked with a mature protocol — one with no uncertainty in data types or message processing. In contrast, for the IPSEC protocol, only one thing was certain at the outset: that the network level data would be subject to some form of cryptographic protection. We set out to track the evolution of the standard through prototyping, both to offer feedback to the designers and to do advance planning on user interface and operational issues that would affect the full security system.

The network security protocol is only one piece of a larger picture: choice and use of data protection transforms, key exchange and management, identification and authentication mechanisms, and application level interfaces.

3 Initial Architecture

At the outset of our work, we had a software architecture for developing and combining network protocol modules [3]. The architecture defines the control paths for message processing in a directed acyclic graph (DAG) of protocol instances. The structure subsumes purely layered approaches and has been used to implement highly efficient protocol structures that are general and semantically rich [8].

Communication functions are based on 5 general operations that are supported by each protocol module instance:

- open session
- close session
- send data
- receive data
- per session option setting

Implicit in the *open* and *close* operations is the notion of a connection between (generally) two parties; the addresses of the parties "name" the connection. Our network software uses a polymorphic addressing scheme, and it is often possible to produce protocols that are oblivious to the exact type of address being used; only the number of bytes in the representation is important.

The other significant support aspect of the infrastructure is a general form of storage management that hides the underlying memory representation from the protocols. Operations for add data to and deleting data from messages are available, and these can translate transparently and efficiently into and out of network byte order [7].

3.1 Simple Network Data Protection

Our initial experiments with adding cryptographic protections to a protocol graph were quite successful in the sense that they required no interface extensions and introduced a new capability into the protocol suites. The facility was to encrypt all network packets leaving a section of the protocol graph (for example, this could include all traffic, only TCP traffic, etc.). This was a limited facility, using only one encryption method and preset keys.

The cryptographic module had four important aspects:

- All outgoing data encrypted
- Typeless
- Stateless
- Crypto algorithm called within "send data", "receive data" routines. The algorithm was wrapped inside the uniform interface for data handling.

The first three aspects meant that the module could be used at almost any part of the protocol graph — adjacent to an application module, at the physical network level, or anywhere in between. The statelessness attribute was a limitation on module placement; with no state information available, DES in CBC mode [2] (or MD5), could not determine where a block of encrypted data began or ended, and unless surrounding protocols preserved block boundaries, decryption could not occur. This property of preserving block boundaries is common to datagram protocols, though, and we could therefore use our new module in conjunction with any datagram protocol.

3.1.1 Types and Modules

The type of a module is determined by its upper types and lower types, as a pair of ordered lists. Protocols must match on type in order to be compatible. For example, TCP requires that upper protocols indicate the TCP port pairs that they use, and it requires that lower protocols use IP addresses.

Our cryptographic modules are written to be polymorphic; each instance is defined by its placement in the graph. Because the modules do no interpretation of addresses, they are type free in design. At runtime, however, each cryptographic module has a place in the graph that defines its type with respect to its upper and lower neighbors; these types must be compatible with the key manager, which inherits its type from the definition of its keys file.

Semi-formally:

$$\begin{aligned}
crypt_{type} &= (upper_{type},\ (transport_{type},\ keymanager_{type})) \\
keymanager_{type} &= (upper_{type},\ transport_{type}) \\
upper_{type} &= (local_{type},\ remote_{type}) \\
local_{type} &= remote_{type}
\end{aligned}$$

Address Types. Here the $uppertype$ is an address type used by protocols "over" (closer to the application) the cryptographic module that is not interpreted by the cryptographic module. The transport type is the address type used by the lower protocol (closer to the network). The key manager determines the appropriate key based on the union of the two types; each key manager module is assigned a type at creation time.

Naming. The $local$ and $remote$ types are ways of naming entities on the local and remote nodes. Most commonly they are Internet address or an ordered list of, for example, TCP port numbers and Internet addresses. Again, semi-formally

$$
\begin{aligned}
local_{type} &= (comp1_{type},\ comp2_{type},\ comp3_{type},\ \ldots) \\
comp[i]_{type} &= an\ atomic\ type
\end{aligned}
$$

(Atomic types are specific to various standards: 48-bit ethernet addresses, 32-bit IPv4 addresses, TCP port numbers, UDP port numbers, etc.)

In practice, we found this to be more complicated than the notation suggests. For example, keys for public key algorithms are based only on the remote address for outgoing data and the local address for incoming data. Signatures use the reverse mechanism. In the case that the local and remote types are lists of components, we found that sometimes we wanted to base key lookup on the first component, sometimes on all components. Finally, the Security Association Identifiers introduced another dimension to the addressing, and we incorporated it as a new address type.

As a result, we now require each cryptographic module to support a set of addressing methods, and these options are configurable at build time and at initialization time. The methods are stated in terms of the address components: asymmetric, first component of local and remote, all components of local and remote, or all components of the security type.

The addition of the encryption was not entirely transparent; it entailed modifying protocols that depended on the pseudo header of the Internetwork Packet protocol (IP). Because DES CBC normally adds padding to a message, in order to bring its length up to a multiple of 8-bytes, the length of the message could change between the transport and network levels. To cope with this we adopted two independent strategies: the "pseudo header fixup" and the length-preserving DES option. The former was a one-line modification to existing protocols, and the latter had an implicit dependency on a minimum data length.

3.2 Transforms

The addressing is a method of associating keys with data streams, but the details of applying the cryptographic algorithm to the message involve a multitude of option possibilities. We will use the term "transform" for the auxiliary data pre- and post-processing that surrounds algorithm application, and the mode of use for the algorithm. During the early stages of development we identified only three useful transform options for DES: cipher block chaining mode (CBC), padding the message data to an 8 byte boundary, and using a technique similar to DES OFB (output feedback) mode to encrypt the bytes beyond the last 8 byte boundary without changing the message length.

Our approach to the state initialization for DES was unique and pleasing for its modular structure: we ignored the DES state altogether. By simply attaching 8 random bytes to the head of the data, prior to encryption, we achieved the goal of obscuring known ciphertext analyses. After decryption, the bytes were discarded. This allowed us to configure "nonce generation" as an algorithm independent part of message processing. Unfortunately, it is not conformant with the usual standards for using DES, so this part of modularity was eventually abandoned with regret.

During the early stages of the standards, there was no consensus on the location or format of auxiliary data; with each round of discussion we added options to encompass the possibilities that might emerge.

4 Adding Policy Enforcement

The next goal was to add selectable cryptographic modules. The major component of the construction was a "virtual protocol" (one which does not alter messages) which acted as a switch. The module used a static table of address pairs mapped to cryptographic algorithm identifiers. All requests to open a connection to the application or transport layers, whether initiated locally or remotely, passed through this module.

This allowed a host administrator to set a policy based on any addressable entity pairs: ethernet addresses, Internet IP addresses, TCP or UDP connections, etc. The cryptographic modules were selectable by the system administrator at build time via the protocol graph. The policy module could be configured at any of several points in the graph. Only the policy table had to be written with knowledge of the addressing types; the policy enforcement was based on the tables and the connection requests.

The type analysis of the policy manager was similar to that for the cryptographic modules, because the implementation was similar. Each policy manager had, in addition to its upper protocols and lower transport protocol, a policy table manager — which was simply an associative lookup table, mapping connection addresses to algorithms, very much like a key manager, which maps addresses to keys.

A variety of configurations were possible, specifying the required algorithm to be used in communicating with remote parties. We could base the keying on ethernet addresses IP addresses, or UDP connections, or TCP connections. Its primary weakness was that only one cryptographic method could be used. We supported DES CBC, a triple DES mode, MD5, RSA, and SHA, but without an interoperable standard for headers, initial values, etc. The configurations could be used by cooperating sites with special purpose environments, but they did not constitute a standard for interoperation.

4.1 Early Application Layer Interface Issues

Despite the limitations of our software, we were able to begin experimenting with application-level uses of it. The modules were instantiated with encryption and MD5 authentications options at the network level (the IP addresses) and a simple header for indicating the algorithm choice was added via a new protocol, our first instance of the IPSEC protocol. The written protocol specifications were still vague, but we decided to move ahead with a minimal definition.

This structure provided enough in the way of "hooks" to let us tie in dynamic Diffie-Hellman key negotiation, a separate line of investigation that proceeded in parallel, far in advance of the specification drafts. Our key manager structure tied together the key negotiation and the cryptographic algorithms neatly, an interface that has remained firm over the course of the research.

Over this structure of network packet protection and dynamic key negotiation, we placed a simple version of the Berkeley remote login program, a program notorious for basing its authentication on insecure IP address information. This allowed us to make minor extensions to rlogin to take advantage of the security information available from the network layer, and to use that information to authenticate IP addresses and to protect passwords [4].

The experience was valuable; it validated our software structure (we added fewer than 20 lines of C code changes to support rlogin), and it indicated the value of network level packet protection; previous work with Kerberos [6], protected data by modifying each application. The network layer scheme seemed to offer similar protection with less effort.

5 Implementing the Standard

As the draft standards took shape, we took on the task of building the scaffolding for the architecture — the parts that were unique to IPSEC. This moved us out of the realm of typeless and composable modules. First, the connection identifiers had to be extended.

The cryptographic attributes associated with a transform were now indicated by an identifying index, known as the SPI (or sometimes SAID). Addressing became more complicated because the combination of destination address and SPI was now essential to identifying the processing parameters for any packet. For this reason we extended the notion of addressing by adding the SPI to the connection identifier. The new address component was implemented as a list of SPI's, extending the types of 3.1.1 so that a connection identifier now requires three components: local, remote, and security.

$$\begin{aligned} upper_{type} &= (local_{type},\ remote_{type},\ security_{type}) \\ security_{type} &= (spi_1,\ spi_2,\ spi_3,\ \ldots) \end{aligned}$$

The "spi" components are all of the same type, the "security association type", which is a number unique for each network host that identifies the cryptographic parameters required for message processing.

The architecture specified no mechanism for associating SPI's with attributes: that is the function of the key management, implemented as a separate protocol. Our key manager modules took on the burden of providing the glue between key management and attribute use by being available for writing (updates, etc.) by the key management protocol, and being available for reading by the cryptographic processing modules. The typeless nature of the keys was a distinct advantage in this design.

5.1 IP and IPSEC

The full IPSEC protocol runs as a subsystem attached to the IP protocol. The semantics of the protocol is more complicated that the garden variety network protocol. A succinct overview of its expected configurations is that it evaluates the regular expression:

$$(I\ A^-\ E^*\ )^*\ P$$

Here I represents IP network layer protocol, A represents the authentication header protocol (AH), E denotes the encapsulation protocol (ESP), and P is some other protocol that will ultimately receive the processed datagram. The $(A-)$ notation means one or no instances of the AH protocol. Each of the A and E instances is identified by a SPI, and they can be independent in terms of algorithm and key. Each of the IP protocol invocations can be independent with respect to sender and receiver addresses [1].

Although a given network node is unlikely to be called up to evaluate more than a few possible values of the regular expression, we did not want to limit our implementation *a priori* to a small set of possibilities. In order to experiment with configurations suitable to encrypting firewalls, MLS gateways, applications with PGP authenticated keys, etc., we needed to have a network layer that was amenable to as wide a set of mechanisms as possible.

The expression represents an abbreviation of the sequence of headers that precede that data. Each header represents a function that is applied to adjacent data.

- The AH header identifies an authentication and integrity processing; it is calculated over the IP header that proceeds it and all the data that follows it.

  I.e., the message structure is $I - A - f(I, A, D) - D$, the concatenation of the IP header (I), the AH header (A), the value of the function taken over the IP and AH headers and the data (D), and the data itself.

- The ESP header identifies encryption processing, and it is applied to all the data that follows it.

  I.e., the message structure is $E - e(H, D, B)$, the concatenation of the ESP header (E) and the encrypted data portion. The encryption is performed over another section of the ESP header (H), the message data (D), and a trailer byte (B). The trailer byte specifies the next protocol; this is how "recursion" through the IP protocol can be indicated.

At this point modularity took a major blow, as we found it necessary to attach two forms of the IP header to the message being processed; the attachment constituted an out-of-band violation of our uniform interface. In addition, the IP protocol endured changes to make it aware of its association with its IPSEC protocol companion and its security parameters.

About 100 lines of changes were necessary in order to develop the interface between the network layer protocol (IP) and the IPSEC modules. The changes involved diverting packets to IPSEC processing, associating IP sessions with IPSEC sessions, and handling the simple header changes. This interface followed the standard interface almost exactly. The major deviations were the attachment of both the host and network form of the final IP header to the message,

[1] Note that a single machine might have more than one IP address.

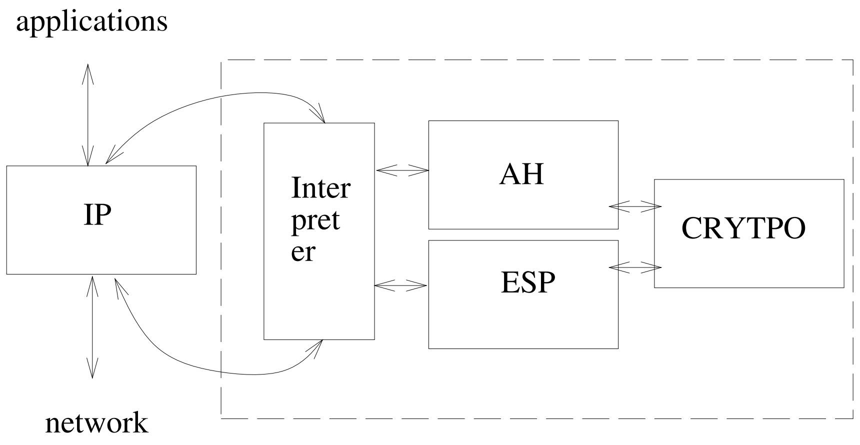

Figure 1: IPSEC module structure

and developing IPSEC as a protocol that is, in essence, connected to IP through a non-standard graph interface — it is neither above nor below ip; it is invoked from IP for both incoming and outgoing packets — not implicitly through the graph data flow structure. [2]

In addition, multiple transforms could be applied to a message, turning the processing into more of an interpreter than a layered data flow. This accounts for the identifier being a list of SPI's rather than a single SPI.

On the other hand, we found a novel way of combining cryptographic modules to achieve a long-standing goal: protocol graphs as subsystems.

5.1.1 Protocol interpreter

To evaluate the expected sequences of security processing for IPSEC, we needed a protocol graph more general than a DAG. The IP protocol needed to be re-entrant, and we needed to ability to handle an arbitrary depth of layering for the security transforms. The resulting module structure with control paths is illustrated in figure 1.

By introducing "interpreter" mode, we were able to solve an architectural problem in protocol composition. Until recently most network systems had fairly static protocol suites with a small number of protocols, although the protocols themselves might support a great number of options. Our architecture followed this model, but we tended to break up protocols into smaller units. We frequently found that we wanted to reuse a unit in a new context, but it was difficult to fit it into the data flow model of a protocol graph. There were two architectural modifications that allowed us to move to a more flexible realm.

Leaf Mode: The first change creates "leaf mode" protocols, which can be called to transform a message and deliver it back to the caller. The name "leaf mode" refers to their isolation in the protocol graph — they have no graph connection to the network and are minima in the DAG. These protocols are stateless and purely datagram oriented; they generate no messages of their own, similar to filters. All our our cryptographic protocols can be used this way.

Loops: The second change allows leaf mode protocol invocations to be controlled by an interpreter, rather than as a side-effect of the graph structure and the uniform interface. The relationship between IP and IPSEC uses this model. Normally, messages pass through IP merely for fragmentation, reassembly, and forwarding. We have modified IP to allow it to utilize IPSEC as a leaf-mode protocol subsystem, and to allow IP to be re-entrant for IP-in-IP encapsulation.

[2] The simple interface does not include all the IP header option processing.

The regular expression for security processing 5.1 lends itself to evaluation in a protocol graph because the AH and ESP protocols are stateless. Composing arbitrary nestings of state-keeping protocols is much more difficult.

This can result in a packet being processed through the network layer as if it had arrived more than once. What is the recipient of such a message to think of it, in terms of authentication and privacy? The concatenation of all parameters provides enough information to query the session state of each component, and we simply note that a most general interface would provide hooks to access this computation "trace". The addressing that lists the security associations for a network session (see 5) gives an application a handle for querying the underlying modules about the security parameters.

5.2 Interoperable Transforms

Beyond handling addressing, the transforms are charged with handling the message manipulations that surround cryptographically processed data. There is a surprising multitude of small changes devised by various network protocol designers:

Prior to the algorithm:

- generating auxiliary data
- initializing the algorithm state from auxiliary data (the IV)
- prepending auxiliary data
- appending auxiliary data
- setting parts of the data to constant or random values
- padding to a fixed length
- encoding the padding length
- other special case handling of padding

After executing the algorithm:

- prepending algorithm state information to the message
- prepending auxiliary information to the message
- removing header bytes
- removing trailer bytes
- writing algorithm results into the data
- adjusting the external measurement of message length (based on padding or other termination method)

Repeating the algorithm (triple DES, etc):

- the mode of each iteration
- interleaving rules
- feedback rules

Each processing option was easily added to our cryptographic modules, and the subsystem for message manipulation was largely responsible for making it easy, because the programmers did not have to worry about buffer boundaries, alignment, and machine dependencies. However, the sheer number of options became a configuration nightmare.

In order to cope with the plethora of options, some of which conflict with one another, we recommend separating the option processing into a module separate from the core algorithm, and we also recommend separating the option selection into yet another module, where the options are grouped according to the standard that they implement (giving RFC number or other relevant document). This allows code re-use without conflict.

6 The value of prototyping

A few months after the IPSEC architectural documents were moved to proposed standards within the IETF, we finished an implementation and prepared it for interoperability testing (not complete at the time of this writing). This is one of, although not the, first implementations of the documents. Our previous prototypes were most useful in achieving this goal.

We believe that software prototypes are valuable as research tools and as a means of getting early validation of standards during their draft stages. Our work with IPSEC and Photuris yielded three results: early identification of performance bottlenecks as a way of focusing implementation research; notice of minimal requirements for supporting site and application level security policies; and extensions to our control structure for processing packets. As a result, we have a structure that we are building on for prototyping other security-related protocols — key exchange, multicast, and routing.

Our work with the network level security involved several prototyping stages over three years. Each stage involved more effort than the previous one, the extra effort being divided between meeting formatting requirements as the standards matured and adding extra functionality. In retrospect one might ask if the extra effort is generally worthwhile. The answer depends on the goals of the project. We offer this guideline based on the stage of the standard:

- Nascent stages of protocol standardization are easy to prototype with generic, reusable software. This software can be used for timing experiments and for simple applications.
- Intermediate stages build on the early ones, but the modules take on specificity in the form of compile-time or build-time options. Experiments with policies, configurations, and application interfaces can begin.
- Late stages introduce specificities that muddy the software and reveal dependencies that were only hinted at in early stages. The modules require additional options that must be coordinated to ensure a functional system. This stage is the most difficult, because early assumptions may be contradicted, ambiguities in the specifications emerge, and some modules may have to be substantially revised.
- The final protocol will be much less modular and flexible than originally envisioned. Software reuse can remain high, but previous experience probably will not predict the number of configuration options that must be introduced.
- Prototyping has the advantage of giving insight into the next stages of protocol design or use. For example, using the prototype in building an application yields insight into interface issues for the protocol implementation in a real system. This foresight assists in keeping the design modular.

7 Conclusions

It is possible to make good use software prototypes during development of a network security standard, even when the progress of the standard is uneven or uncertain. Having a strong software architecture to begin with is a major advantage, but being locked into a fixed control structure could be an impediment. Security processing involves coordination between several components of network software: applications, cryptography, key management, data transport, and auxiliary functions for attaching security attributes to network sessions. A strongly modular system greatly promotes development of re-usable software.

References

[1] R. Atkinson. Security architecture for the internet protocol. RFC 1825, 1826, 1827, August 1995.

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

[3] 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.

[4] G. Kim, H. Orman, and S. O'Malley. Rlogin and an example of a composable security policy. In *Proceedings of the 1995 Usenix Security Symposium*, June 1995.

[5] P. Metzger and W. Simpson. Encryption and authentication transforms for the internet protocol. RFC 1828, 1829, 1851, August 1995.

[6] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Salzer. Kerberos authentication and authorization system. Technical Report Project Athena Technical Plan, Section E.2.1, Massachusetts Institute of Technology, April 1987.

[7] S. O'Malley, T. Proebsting, and A. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, Aug 1994.

[8] H. Orman, E. M. III, S. O'Malley, and L. Peterson. A fast and general implementation of mach ipc in a network. In *Proceedings of the 3rd Usenix Mach Conference*, pages 75–88, Apr 1993.

[9] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. Paving the road to network security, or the value of small cobblestones. In *Proceedings of the 1994 Internet Society Symposium on Network and Distributed System Security*, Feb 1994.

[10] W. Simpson and P. Karn. The photuris session key management protocol. ietf draft, ftp from nic.merit.edu/documents/internet-drafts/draft-ietf-ipsec-photuris-06.txt, August 1995.