Design and Implementation of Modular Key Management Protocol and IP Secure Tunnel on AIX

Pau-Chen Cheng, Juan A. Garay, Amir Herzberg, and Hugo Krawczyk
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

Design and Implementation of Modular Key Management Protocol and IP Secure Tunnel on AIX

Pau–Chen Cheng Juan A. Garay Amir Herzberg Hugo Krawczyk

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.

{pau,garay,amir,hugo}@watson.ibm.com

April 28, 1995

Abstract

*This paper presents the design principles, architecture, implementation and performance of our modular key management protocol (*MKMP*) and an IP secure tunnel protocol (*IPST*) which protects the secrecy and integrity of IP datagrams using cryptographic functions. To use the existing IP infrastructure, MKMP is built on top of* UDP *and the IPST protocol is built by encapsulating IP datagrams.*

1 Introduction

As Internet evolves from an academic/research network into a commercial network, more and more organizations/individuals will connect their internal networks/computers to Internet. Secrecy and integrity of data transmitted over the *insecure* Internet have become a primary concern and cryptographic data encryption and authentication constitute the tools to address this concern.

In this paper we describe an architecture and its implementation that provides for secure communication over the currently insecure Internet. This architecture includes protocols for *key management* and a *secure tunnel* mechanism that provide network–layer packet encryption and/or authentication. These protocols enhance systems supporting TCP/IP, firewalls, secure tele-commuting, secure mobile devices, etc. At the heart of the security architecture is a set of protocols that we have designed for the management of cryptographic keys as required for the establishment and maintenance of *security associations*. A security association between two communicating systems represents the information shared by these systems in order to control the secure communication between them. This information includes secret keys, key lifetimes, nonces, crypto algorithms, parameters, etc..

We subscribe to the view that IP–layer [Pos81] is a good place to secure data. The reasons include:

1. The secrecy and integrity of data can be protected in an *internetwork environment* without affecting higher–layer protocols and applications;

2. The integrity of IP headers can be protected using cryptographic techniques and therefore *packet filtering* [DBC92] can be done based on *authentic* information. This property is very useful for Internet firewalls [CB94].

There are many possible applications for encryption and authentication between two systems on the Internet. Two examples are (refer to Figure 1):

1. **authenticated secure communication** *across* **insecure domain** : Host to host (similarly, network to network): encryption and authentication between the two systems allows the two hosts (networks) to have secure communication through the insecure Internet;

2. **authenticated secure communication** *from* **insecure domain** : If a firewall–protected network communicates with an external system, then encryption and authentication between the system and the firewall allows for secure communication through the insecure Internet. This scenario is useful, for example, in tele–commuting and connecting mobile users/computers to their bases.

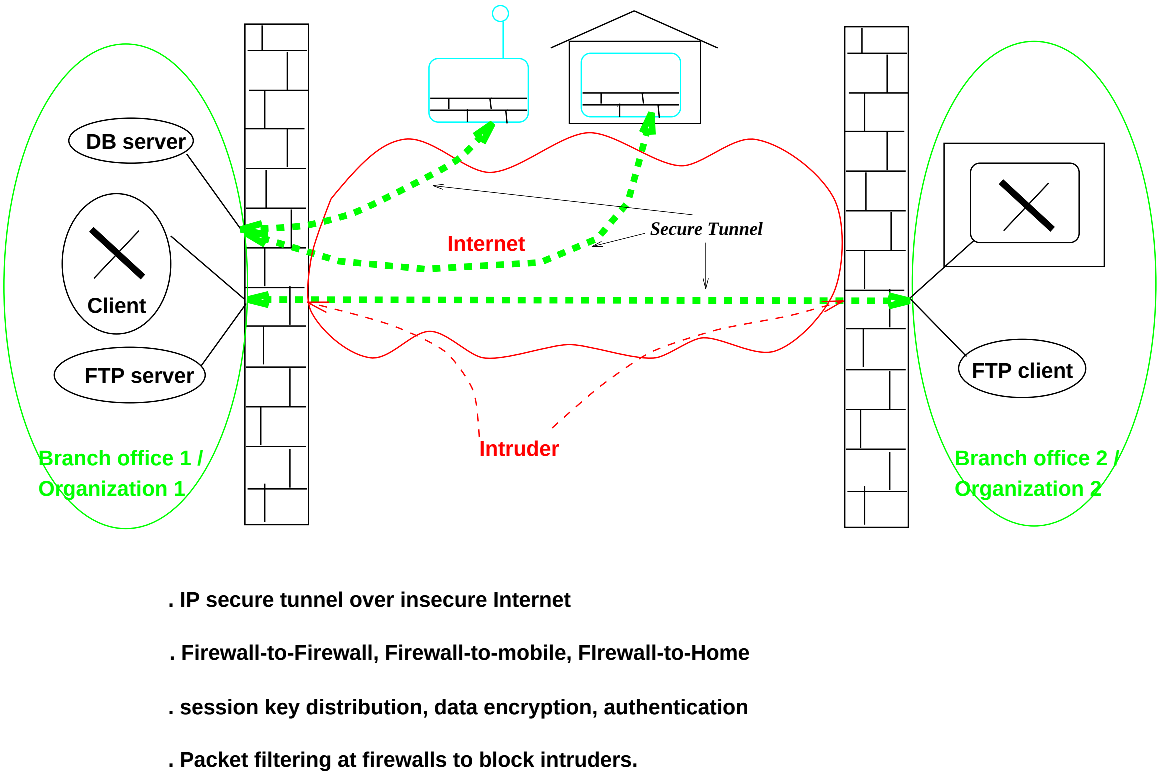

Figure 1: Applications of IPST

At the heart of our security architecture are the *IP Secure Tunnel Protocol (IPST)* and the *Modular Key Management Protocol (MKMP)*. IPST follows the spirit of discussions in the IETF IPSEC (IP Security) working group. It is an encapsulation protocol, namely, one that defines the format of an IP packet which encapsulates another IP packet. The encapsulated packet may be encrypted and/or have cryptographic integrity protection. An IPST packet, including the encapsulated IP packet, is placed in another IP packet and transmitted over the Internet. *MKMP* is a (set of) protocol(s) we have designed for the management of cryptographic keys as required for the management of security associations in IPST. It provides secure mechanisms for periodic refreshment of keys and derivation of working keys as required for the multiple cryptographic functions used with a single security association.

We have prototyped our protocols, which will be part of the new release of IBM's "NetSP Secure Network Gateway" (firewall) product. In addition, we are proposing our key management approach to the IETF IPSEC Working Group for possible inclusion as part of the Internet standard.

Our work differs from *swIPe* [IB94a, IB94b] in that : 1) a key management protocol is implemented and linked with IPST; and 2) the IPST function is placed inside the kernel IP module and not in a network device driver.

The organization of this paper is as follows. Section 2 describes the *MKMP* and the IPST protocols, sections 3 and 4 describe the architecture of our implementation and section 5 discusses its performance.

2 Protocols

This section presents the design philosophy and high-level description of the *Modular Key Management Protocol* [CGHK94] and the *IPST* protocol implemented by us.

2.1 Modular Key Management Protocol

A typical key management scheme will have two main phases. One in which a "master" key is shared between the communicating parties, and the second, in which the already shared master key is used for derivation, sharing and/or refreshment of additional session keys [1] to be applied in the cryptographic

[1] the term "session key" is used here to denote short-lived keys as opposed to long-lived master keys; it does not imply

transformations. The split into these two phases is not mandatory, and in fact there are systems that do not establish (at least explicitly) this separation. However, we argue here that for most scenarios - and IPST is one of them - this explicit separation has a significant methodological and design value. In particular, we advocate the separation of two modules: one for the sharing of the master key, and one for the key management "below" the shared master key. Thus, our approach to key management is *hierarchical*, namely, session keys are derived from the shared master keys. Master keys are derived using any of the well-established methods like public key, key distribution centers, or manual key installation. The approach is illustrated in Figure 2.

In this paper we provide a high-level description of the basic mechanisms for the management of *session keys* (the "lower" module). Our protocol has natural extensions for the derivation of master keys from public keys; the description of these particular mechanisms is beyond the scope of this paper. Very importantly, the session key protocol can also be combined with any other mechanism for master key agreement, such as key-distribution centers (e.g., Kerberos), manual key installation, etc.

2.1.1 Session key negotiation protocol

The basic goal of a key negotiation protocol is to provide both intervening IP nodes with a shared *session* key. The key is then used for data authentication and encryption, thus allowing the establishment of a secure tunnel between the two parties. A basic assumption that we make is that parties are (usually) able to establish periodic interactive communications (as opposed to just sending information in a one-way mode). Interactive key refreshment has the advantage of providing keys that are fresh and independent from the past session keys (the only dependence is to the current shared master key). The cryptographic handshake serves also for direct authentication between the parties.

Another important goal of the protocol is efficiency, namely, to keep both the number of messages between the parties and the computational overhead (e.g., the number of expensive public key operations) to a minimum. Our session key protocol requires two flows and no exponentiations at all if the parties already share a master key (in which case only efficient symmetric-key techniques are used). It is worth noticing that by having a highly efficient method for session key renewal, the need for frequent master key update, which is usually computationally intensive, is alleviated. The third consideration is the level of security provided by the protocol. Our approach guarantees a basic security principle for session keys, namely, even if an eavesdropper is eventually able to derive the key for one session, then future session (and, of course, master) keys are not compromised. This follows from properties of pseudorandom functions, and our particular application of these functions.

or require a session-based communication model.

Finally, simplicity and being amenable to analysis and proof are important features of any cryptographic protocol. The protocol presented here is structured, simple, and thus easier to analyze. Indeed, methods similar to those of [BGH+93, BR93] can be used to establish the protocol's desired security properties.

Figure 3 shows the "cryptographic skeleton" of the session key negotiation protocol, including only the relevant information. There are two parties, S (for sender) and R (the receiver). S is the party that initiates the protocol. We use the following terminology:

N_X: A nonce (i.e., a random number) chosen by X.

K: The shared master key.

MAC_K: A Message Authentication Code (or integrity check function) which is applied to a piece of information for authentication using a secret key K. (Examples include block ciphers, e.g. DES, in CBC-MAC mode [Ass86], or key-ed cryptographic hash functions, e.g. keyed-MD5 with prefixed and/or suffixed key [Tsu92].)

SK: The session key, outcome of the protocol.

f_K: a pseudorandom function with index K. (Roughly speaking, pseudorandom functions are characterized by the pseudorandomness of their output, namely, each bit in the output of the function is unpredictable if K is unknown.) The above examples of MAC functions are also believed to be pseudorandom functions.

We now turn to the process of establishing a session key between S and R. (The same protocol allows for periodic key refreshments within a session). This includes mutual authentication and the exchange of SK, the session key. It is assumed that S and R already share a master key K, as well as the nonce N_R (exchanged in a previous run of the protocol). The nonce serves as a challenge for guaranteeing the freshness of the authentication (i.e., to avoid *replay* attacks). Keeping a shared nonce between sessions is not essential: it can be replaced by use of time stamps (at the expense of requiring good clock synchronization) or by adding an extra flow to the protocol (at

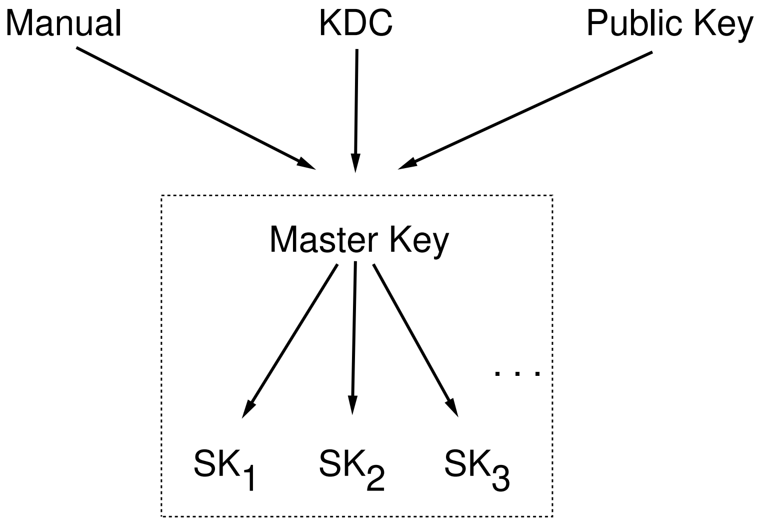

Figure 2: Hierarchical approach to key management.

the expense of performance). The nonce also serves the purpose of alleviating the effect of the *clogging* (denial of service) attack. In any case, the nonces do not require any secrecy, i.e., they can be transmitted in the clear.

Notice that the session key SK is not explicitly transmitted. This avoids the need to encrypt this key as well as the need to authenticate it. The authenticity and freshness of SK are derived from the authenticity and freshness of the expression T. (Even if an adversary succeeds in replaying an old message from S to R, e.g., in case a time-stamp is used instead of the nonce N_S, the freshness of SK is guaranteed by the incorporation of N'_R, chosen by R, into the MAC expression T from which SK is derived.)

Finally, we stress that usually one rquires more than one key for a given security association. For example, one needs different keys for encryption and authentication of information, and in some cases the keys are used uni-directionally. To derive more than one key is straightforward: instead of a single application of $f_K(T)$ one applies $f_K($"transform-id"$, T)$ for each required key, where "transform-id" is a unique identifier that identifies the algorithm for which the key is to be used (e.g., DES-CBC), the key length, the direction (e.g., for message authentication from S to R only), etc.

2.2 IP Secure Tunnel Protocol

In order to use existing IP infrastructure, the IP Secure Tunnel Protocol is an encapsulation protocol.

Figure 4 shows the packet format after encapsulation. The IPST payload is the to–be–protected IP datagram. If the payload's secrecy is to be protected, then it is encrypted before being put inside the IPST packet. If the payload's integrity is to be protected, then a message authentication code is computed on the concatenation of the IPST header and the payload and is appended to the payload. We stress, as an important principle followed by our design, the independence of the network-layer protocol from the key management module. Indeed, the only interface to key management is provided via the *security association ID* (SAID) that serves as a pointer to the key and other attributes of the secure association (i.e., the particular tunnel established between the sender and recipient of the packet).

The IPST packet header includes the following fields:

version : version number of IPST.

protocol : protocol number of IPST payload[2].

IPST header length : number of 4–byte words in IPST header. The IPST header must be padded to an integral multiple of 4 bytes.

flags : a bit vector indicate what operation(s) (encryption/authentication) is(are) performed on the IPST packet.

IPST packet length : number of bytes in the IPST packet, including IPST header.

[2] Our implementation only supports IP payload. But there is no reason for our IPST not be able to support payloads of other protocols.

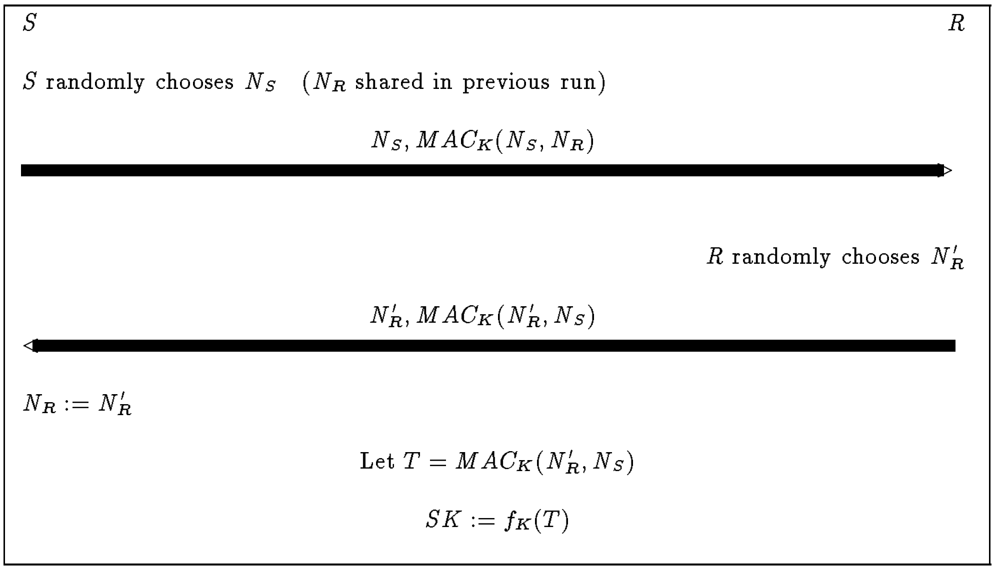

Figure 3: *MKMP*: The session key negotiation protocol.

SAID : Secure Association ID of the IP secure tunnel through which the payload is put. A basic property of SAID's is that they are freely chosen by the local implementation; no special structure or semantics are assumed. More on SAID can be found in [Atk95].

STPI : Secure Transformation Prepended Information. The content and length of this field depends on the particular cryptographic operations performed on the IPST packet. For example, if DES–CBC is used for encryption, then this field will contain the CBC initial vector.

Each secure association has two keys, one for encryption and one for message authentication. The use of different keys is to avoid possible crypto–coupling between encryption and message authentication. Our prototype uses DES–CBC for encryption and keyed–MD5 (key is pre–fixed and post–fixed) for message authentication. The combination of prepended and appended key for keyed-MD5 is chosen for added security of the authentication function (for security discussions of keyed-MD5 see [Tsu92, BCK95]). The message authentication computation is done prior to encrypting the payload. While performing the authentication on the ciphertext (i.e., after encryption) has the advantage of saving the decryption operation in case of a bogus message, authenticating the plaintext (i.e., before encryption) gives assurance of correct decryption for the recipient (decryption errors can be produced by use of the wrong key, etc.). Although our prototype only supports DES–CBC and keyed-MD5 at present, our crypto functions are implemented as plug–in, replaceable modules and are not bound by DES–CBC and MD5 (or the order of encryption/authentication). More on crypto functions and their implementation is described in section 3.

3 System Architecture

Figure 5 shows the system architecture of our implementation on AIX 3.2.5[3]. It consists of five major components : *Modular Key Management Protocol* (MKMP) engine, *policy* engine, *IP engine*, *IPST* engine, and *crypto* engine. There are also some *glue* components : *tunnel interface* and *tunnel cache* to link the IPST engine and the MKMP engine, and *policy interface* and *policy cache* to link the policy engine and the IP engine. The policy engine and the MKMP engine run as separate processes in user space, other components run in the kernel.

3.1 MKMP Engine and Tunnel Cache/Interface

The MKMP engine establishes and manages security associations; it is divides into 2 separate processes, the session key engine and the master key engine. The master key engines negotiate the master keys

[3]IBM's version of UNIX for the RS/6000 processor family

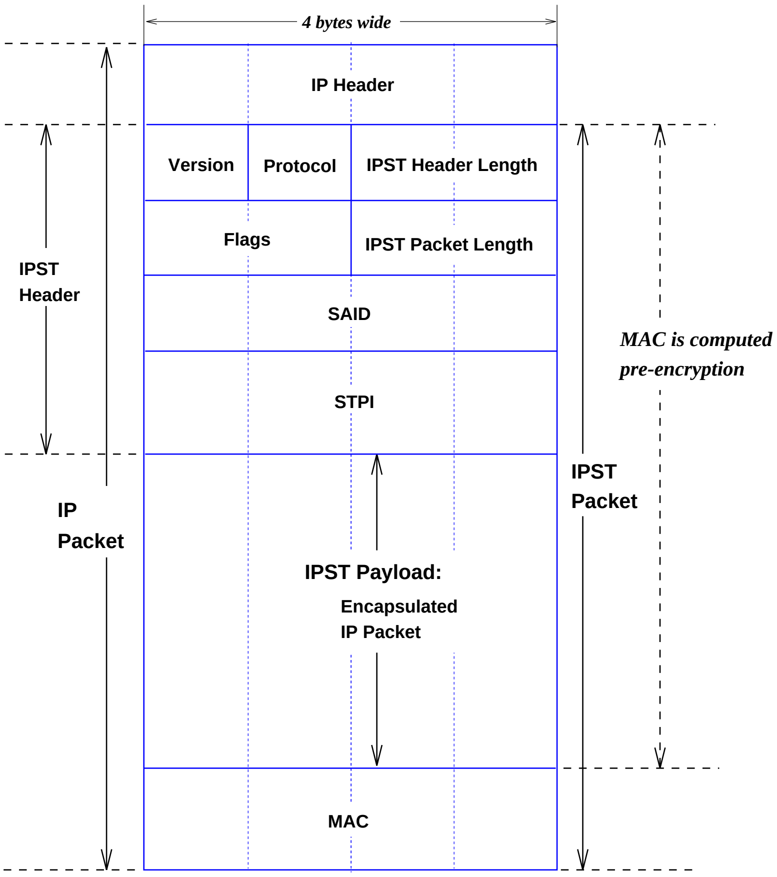

Figure 4: IPST packet format

and security association parameters, including crypto algorithms and parameters to be used with session keys, sizes of session keys, key lifetimes, key refreshment policy, etc., and pass these information to the session key engines. The session key engine is a process which implements the session key protocol described in section 2.1; it accepts master keys and security association parameters from master key engines and uses these information to establish security associations with other session key engines. The session key engine caches security associations in the tunnel cache, a kernel extension, through the tunnel interface, a pseudo device driver. The master key engine may be a simple user-level command which implements manual distribution of master keys. Or, it may use a KDC[4]–based protocol, such as Kerberos or NetSP [BGH+95]. Or, it may be a process which derives master keys from public keys. At present, the master key engine implements manual key distribution and parameter negotiation. We are currently implementing a master key engine based on Diffie–Hellman key exchange [DH76] to be described in a forthcoming paper. More on MKMP engine is described in section 4.

[4] Key Distribution Center

3.2 Policy Engine/Cache/Interface

The policy engine is implemented as a user–level command. Its job is to translate human–understandable IPST policy specifications into an internal representation in the form of a *list*. The first entry in the list that matches an IP datagram determines how the datagram should be handled. The list is cached in the policy cache, a kernel extension, through the policy interface, a pseudo device driver. Each entry in the list has the format shown in Figure 6.

An entry specifies, based on an IP datagram's source/destination addresses (masked with src_addr_mask/dest_addr_mask), protocol, source and destination port numbers, whether the datagram should be encapsulated (enc/mac); and if encapsulation is necessary, whether the bigger IP datagram should have different source (im_src) and/or destination addresses (im_dest). The boolean field *enc/mac* specifies whether encryption and/or

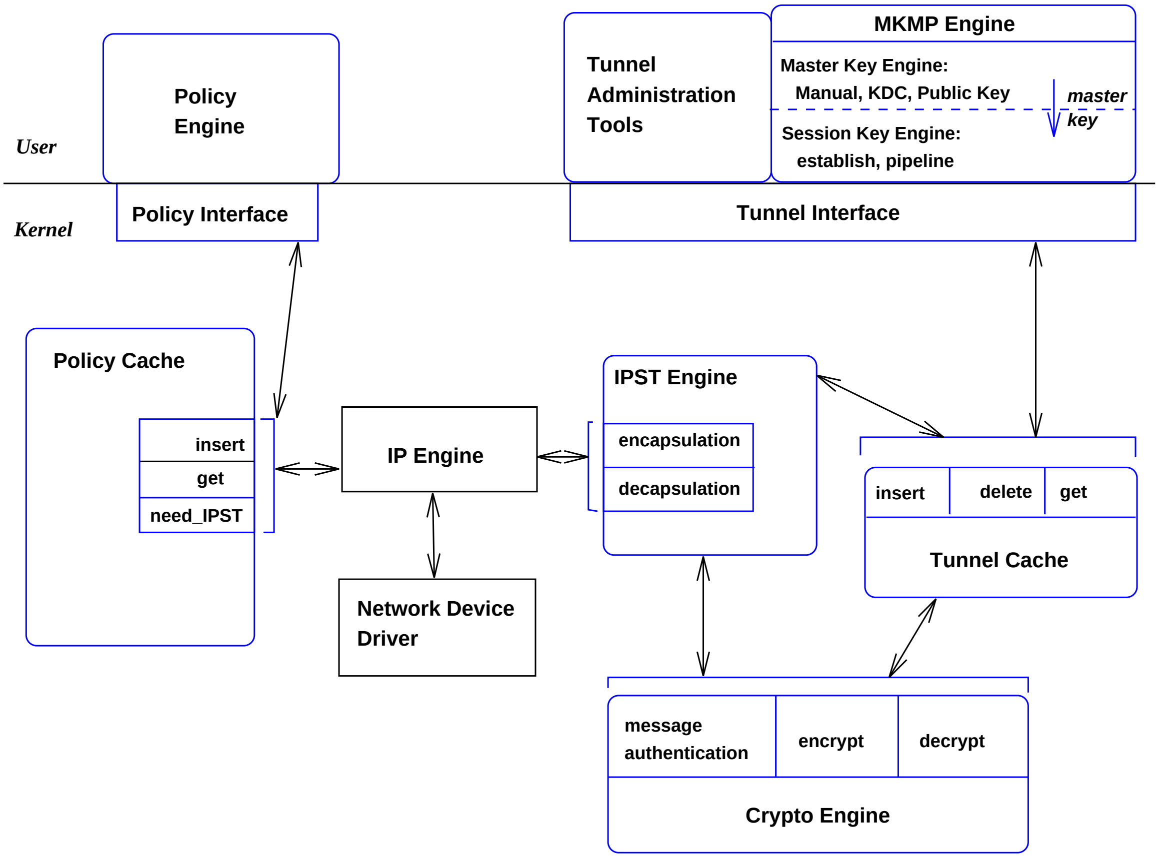

Figure 5: IPST System Architecture

src_addr:src_addr_mask:dest_addr:dest_addr_mask:im_src:im_dest:protocol:src_port:dest_port:enc/mac:master_key_ctx_id

Figure 6: IPST policy entry format

message-authentication is required; if neither is required then no encapsulation is performed. The *im_src* and *im_dest*[5] are intended for secure tunnels between two firewalls; they are IP addresses of the two firewalls (refer to Figure 1). The use of *master_key_ctx_id* (master key context ID) is explained in section 4. Except of the *enc/mac* field, all the other fields can be *wild cards*.

Besides IPST policy, the policy engine also translates human–understandable *input* and *output filtering* policy specifications into two separate lists and caches the lists in the policy cache. The entry format of these lists is the same as that of the IPST policy. For each received IP datagram, the IP engine first decapsulates the IP datagram if necessary; the IP engine then passes the (decapsulated) datagram, and the way it was protected (encrypted/authenticated or none) to the policy engine.

The policy engine matches the datagram's addresses, protocol and port numbers against the input filtering policy list. If a match is found, then the enc/mac and master_key_ctx_id[6] fields determines the minimal protection the datagram must have had. If the datagram had the required protection, the policy engine returns an OK to the IP engine. If no match is found or the datagram did not have the required protection, the datagram is discarded. Latter, if the datagram is to be forwarded, the IP engine will invokes the policy engine to match the datagram against the output filtering policy list.

The policy engine and policy cache are implemented as a *black box* relative to the IP engine. In other words, the IP engine only asks whether a datagram should be encapsulated/received/forwarded and in what way but does not care how the answer

[5] "im" means *intermediate*.

[6] This field specifies required crypto algorithms used on the datagram, see section 4.

is reached. This design choice is less for code modularity but more for flexibility on how policies can be specified. In other words, the policy engine and policy cache can be replaced to fit a special need without affecting other components.

3.3 IP/IPST/Crypto Engines

The IP engine is a modified version of AIX kernel IP module. For each IP datagram to be transmitted, the IP engine queries the policy cache to determine if the datagram should be encapsulated, and if so, in what way. The IPST engine is invoked accordingly to perform the required encapsulation on the datagram. The output of the IPST engine is a new, larger IP datagram encapsulating the original one (see section 2.2). The IP engine then sends the larger IP datagram in the usual way; the larger IP datagram my be *fragmented* because of added headers and message authentication code. For each received IP datagram, the IP engine checks if the IP datagram contains another encapsulated IP datagram and invokes the IPST engine to perform decapsulation if necessary. A decapsulated IP datagram is first filtered according to the input filtering policy (refer to section 3.2). If the datagram passes the filter, it is processed in the usual way. Otherwise, it is discarded. *Since all IPST processing happens in the IP layer, no other kernel protocol modules (TCP, UDP, ICMP, etc.), user–level applications (FTP, TELNET, rlogin, X–window, etc.), nor network device drivers are changed.*

The IPST engine invokes the crypto engine to perform encryption/decryption and message authentication. The crypto engine gets the necessary keys from the tunnel cache. Unlike *swIPe* [IB94a, IB94b], the IPST engine is an AIX kernel extension and not a pseudo network device driver. This design choice is based on the following reasons :

- Making IPST engine a pseudo network device driver may impose a strong and undesirable coupling between network security policies/operations and routing policies/operations. We believe in following modular, layered design principles whenever possible.
- Some subsystems, like NFS, have their own (partial) IP engines for performance or other reasons. For example, an NFS server may cache the IP datagram sent to clients; encrypting these datagrams in a network device driver may interfere with the caching mechanism. Making IPST engine a kernel extension enables it to be reused by these subsystems in proper ways.

The crypto engine maintains two separate lists of *crypto systems*, one for encryption and one for authentication, each with a generic interface. Each crypto system is a collection of cryptographic functions and related parameters, and is assigned a unique ID. For example, DES-CBC and keyed MD5 are each crypto systems. Crypto systems are implemented as loadable kernel extension modules which register with the crypto engine when they are loaded into the kernel. This design idea comes from Kerberos version 5 [KN93] code; it enables crypto algorithms to be implemented as *replaceable plug–in modules.*

Crypto systems are passed chains of kernel memory buffers (*mbuf's* [LJFK86]) containing payloads on which they operate, and return their results in new chains of kernel memory buffers. That is, *crypto systems do not operate on payloads in place.*

4 Architecture of Key Management Engines

This section describes the implementation of the modular key management protocols described in section 2.1. A high–level description of the implementation is given in section 3.1; this section gives more details. Figure 7 shows the architecture of the session key engine and the master key engine and their relations with other parts of the OS. The design goal of the architecture is modularity, flexibility and portability. Since we envision that the session key engine may interoperate with many master key engines of different types, the session key engine is an *independent process separated from master key engines.* A master key engine sends to the session key engine a UDP message containing a *master key context* through a well–known port to initiate the session key protocol between two systems. Figure 8 shows the master key contexts in a session key engine. A master key context contains *ID of the context*, the value of the master key, ID and lifetime of the master key, ID's of the local and remote systems, the shared nonce N_R, security association parameters and *session key refreshment factor.* The master key is actually *a pair of keys*, one is used to authenticate session key protocol messages and the other is used as an input to the pseudo–random function to derive session keys. A run of the session key protocol derives two session keys, one for IP datagram encryption and the other for authentication. The ID's of the encryption and authentication algorithms (the "transform–id" in section 2.1) are used to index the pseudo–random functions to generate different keys. A session key refreshment factor determines when, in the lifetime

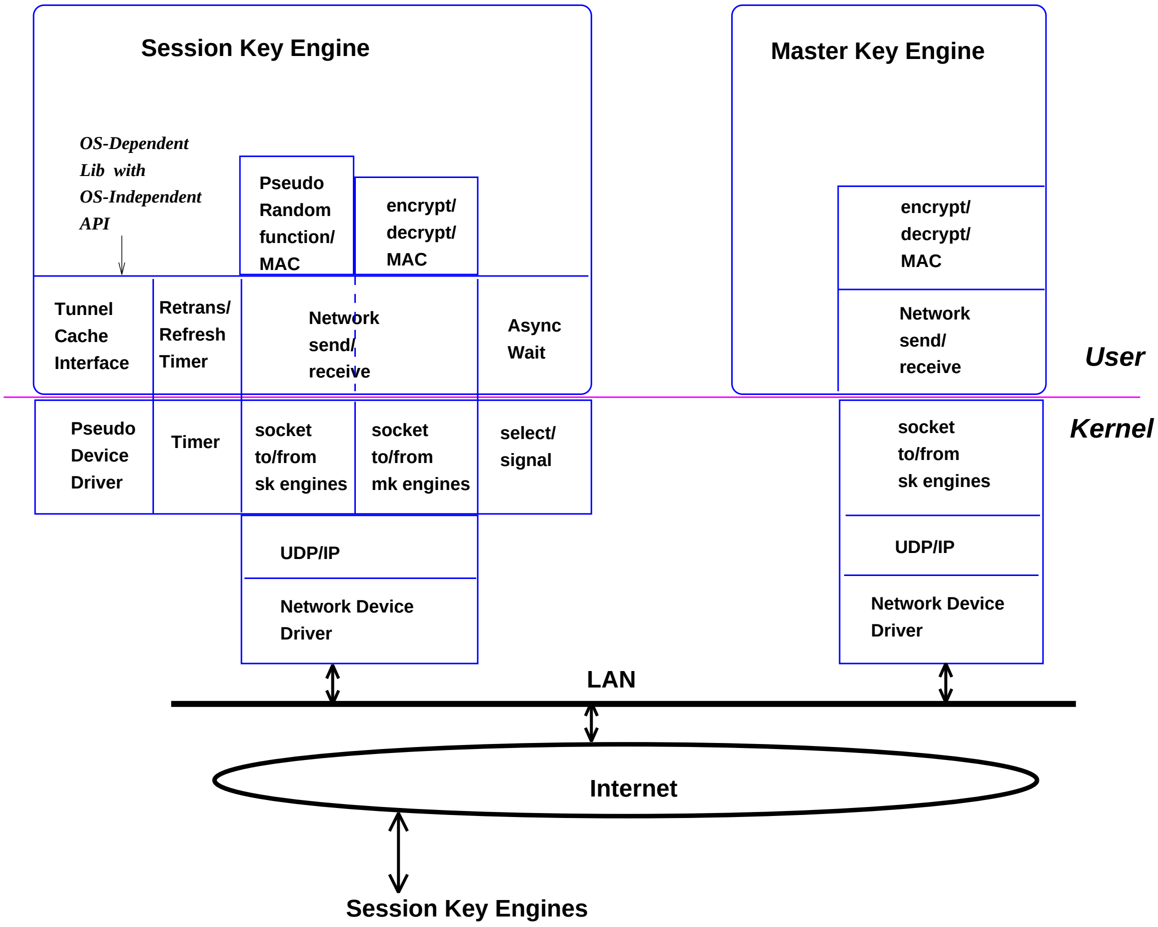

Figure 7: Architecture of Key Management Engines

of a session key, should a key refreshment be started to get the next session key. If the lifetime is 60 minutes and the key refreshment factor is 10, then the key refreshment should be started when there is 6 (60/10) minutes remaining in the lifetime. All the information in a master key context should be negotiated by two master key engines (including human beings if manual master key distribution is used). To protect the communication between a session key engine and a master key engine, all master key contexts in messages are encrypted and authenticated. A secret key, read from a file, is shared by the session key engine and the master key engine for this purpose. Note that this design allows a session key engine and a master key engine to run on different systems.

Two session key engines on two different systems communicate using UDP messages through a well–known port. The *skeleton* of these UDP messages are described in section 2.1; its format is shown in Figure 9 :

- **type**: type of the protocol.
- **version**: version of the protocol.
- **flags**: bit vector indicating whether the message is an S, R or ACK message (see explanation below).
- **length**: length of the message in words (4–byte).
- **source IP address**: IP address of the sender of the message.
- **destination IP address**: IP address of the intended receiver of the message.
- **master key ID**: see explanation above.
- **SAID**: ID of the security association to–be–established by the run of the session key protocol.
- $\mathbf{N}_S$, $\mathbf{N}_R$ and **MAC**: refer to section section 2.1.

A session key exchange and its future key refreshments happen within the scope of their master key

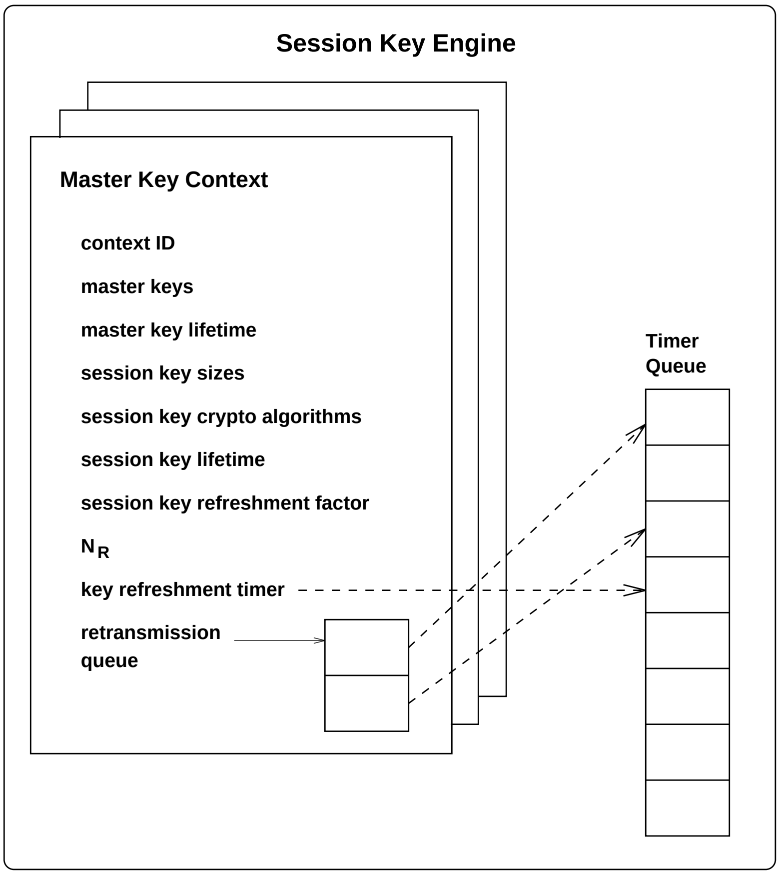

Figure 8: Master Key Contex in a Session Key Engine

context. Because UDP messages may be lost, message retransmission is provided by the session key engine. To prevent endless retransmission, we add a third ACK message from S to R to acknowledge the second message from R to S. A master key context stored in a session key engine keeps *a queue of messages for retransmission* (see Figure 8), and timers for message retransmission and key refreshment/deletion[7]. The messages on the retransmission queue also serves as storage of protocol state information. For each key refreshment, the session key engine creates a new security association and assigns a new SAID to the new association. The session keys in this association are new, but other parameters remain the same.

A security association is always cached with the ID of its master key context in the tunnel cache. This ID allows an IPST policy to specify a fixed set of security parameters, such as crypto algorithms, session key size/lifetime, etc., while the session keys are being frequently refreshed (see the master_key_ctx_id field in Figure 6).

To provide portability across different platforms, we implemented a layer of OS–dependency library which provide OS–independent API's to provide the following services:

- secure communication : for network communication, encryption and authentication of messages from the master key engine to the session key engine.
- timer/alarm : for retransmission and key refreshment/deletion.
- asynchronous wait : for capturing asynchronous events (e.g, time–out events and receipts of messages).
- tunnel caching : for caching security associations.

5 Performance

All our tests and performance measurements are done between two RS/6000 systems running AIX 3.2.5 and

[7] A key is deleted after its lifetime expired. We allow a short *grace period* before a key's deletion.

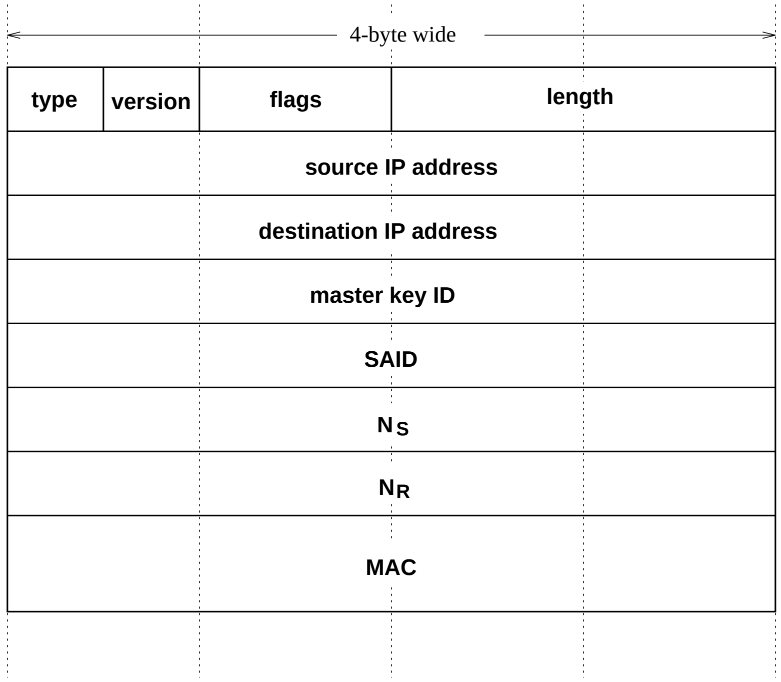

Figure 9: Format of Session Key Message

connected by a 16M-bit token ring network. One system is a model 360 with a 50MHz cpu and 64 Mbytes of RAM. The other system is a model 530 with a 25MHz cpu and 32 Mbytes of RAM.

5.1 Performance of IPST

Our measurements show, as expected, that DES–CBC encryption and decryption operations contribute the most to performance degradation and keyed–MD5 has less effect on performance. We use optimized IBM DES–CBC code, modified by us to work with the *mbuf* data structure. The MD5 code is from RFC1321 [Riv92]. The raw performance figures of these codes running on our test systems are shown in Table 1. These figures are derived by using a C language *for loop* to perform DES/MD5 operations on 16384 bytes of data 400 times and then taking average over elapsed time; i.e., $performance = 16384\ bytes \times 400/(elapsed\ time)$. The compiler we use is the *xlc* C compiler with optimization (-O) turned on.

To benchmark the performance of our IPST implementation, we first established the tunnel by running the MKMP session key protocol to exchange keys and then measured the speed of *ftping* the `/unix` file (1637353 bytes) from Model 360 to Model 530 in different conditions. The results are shown in Table 2. The degradation comes from en/de–cryption, message authentication and IP fragmentations caused by the added packet headers.

We also tested the performance of interactive applications to check its acceptability to human users. To test this, we ran a telnet session (with vi, ls -l, etc.), an X–window version of the AIX SMIT command and an MPEG–II player. The servers (including X server) were on model 360 and the clients were on model 530. DES–CBC and keyed–MD5 were all turned on. We did not measure the actual performance figures but the response time was acceptable; there were barely noticeable delays when scrolling the screen during a vi session.

Figures 10 and 11 shows the IP output and input processing times versus *pre–encapsulation/post–decapsulation* datagram size measured on model 360[8]. Each figure shows 5 curves: total processing time (including encapsulation/decapsulation time), total encapsulation/decapsulation time (including encryption/decryption + MAC time), encryption/decryption + MAC time, encryption/decryption Time, IP fragmentation processing time. It is worthwhile noticing that :

- Encryption/decryption time is the dominant factor in performance degradation.
- The difference between total processing time and total encapsulation/decapsulation time is the

[8]Its SpecInt92 is 57.5.

	Model 360 50 MHz Mbits/sec	Model 530 25 MHz Mbits/sec
DES–CBC	4.3	3.7
keyed–MD5	25.6	17.1

Table 1: Raw Performance Figures of Crypto Operations

	ftp /unix Mbits/sec	Degradation Ratio
vanilla IP	8.1	1:1
DES–CBC + keyed–MD5	1.6	5:1
DES–CBC only	2.0	4:1
keyed–MD5 only	3.7	2.2:1

Table 2: IPST Performance Benchmark

vanilla IP processing time. This difference is almost constant because vanilla IP processing only looks at IP header which is almost always 20–byte long.

- Performance degradation ratio is not constant but roughly proportional to the IP datagram size. This is because IPST processing looks at the entire datagram to do en/de–cryption and message authentication. For interactive applications which send small datagrams, the degradation is small.
- IP fragmentation does affect performance; but it does not happen until the datagram size is larger than 1436 bytes[9]. At this size, the effect of fragmentation is trivial compared to the effect of en/de–cryption.

5.2 Performance of MKMP

Our tests showed that our session key engine can sustain a few key–refreshments per second. Te see the effect of key refreshments on applications' performance, we ran simultaneous telnet, SMIT and MPEG–II player sessions between two test systems with DES–CBC and keyed–MD5 turned on under different key–refreshment period. Starting with 60 minutes, the refreshment period is cut in half repeatedly. An observable performance difference happened when the period is down to 2 seconds. We believe improvement in the synchronization mechanism between the read and insertion operations into the tunnel cache can improve the performance. Further experiments are being conducted.

[9] The *mtu* [LJFK86] is 1492 bytes on model 360.

Availability

For availability of more detailed design information and source/binary codes, please contact the authors by e–mail.

Acknowledgement

We wish to thank our IBM colleagues for their many useful technical advices and logistics support; this work would have been impossible without their help. Specifically, we wish to thank Erol Basturk, Mihir Bellare, Maria A. Butrico, Chee–Seng Chow, Mark C. Davis, Edie E. Gunter, Donald B. Johnson, Dilip D. Kandlur, Jed Kaplan, Arvind Krishna, Mark H. Linehan, Charles C. Palmer, Ed Pring, Stephen E. Smith and Moti M. Yung.

References

[Ass86] American Bankers Association. *American National Standard for Financial Institution Message Authentication (Wholesale).* ANSI X9.9, 1981, revised 1986.

[Atk95] Randall Atkinson. IPv6 Security Architecture. IETF draft-ietf-ipngwg-sec-00.txt, February 1995.

[BCK95] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying MD5 – Message Authentication via Iterated Pseudorandom-

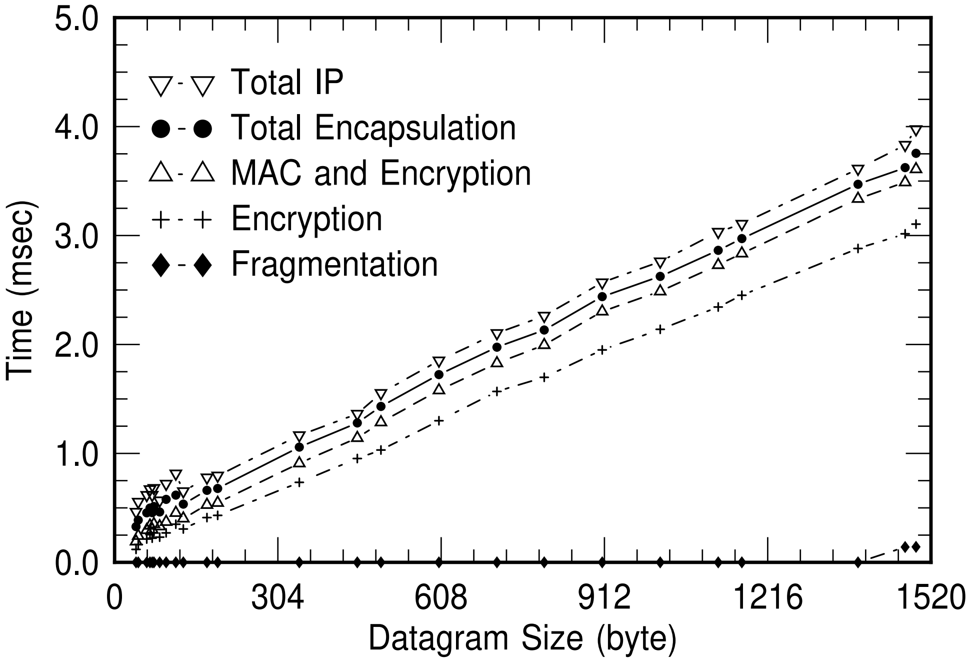

Figure 10: IP/IPST output processing time vs. datagram size on 50–MHz RS/6000 model 360

ness. paper in preparation, February 1995.

[BGH+93] Ray Bird, Inder Gopal, Amir Herzberg, Philippe A. Jason, Shay Kutten, Refik Molva, and Moti Yung. Systematic Design of a Family of Attack–Resistant Authentication Protocols. *IEEE Journal on Selected Areas in Communications*, 11(5), June 1993.

[BGH+95] Ray Bird, Inder Gopal, Amir Herzberg, Phil Jason, Shay Kutten, Refik Molva, and Moti Yung. The KryptoKnight Family of Light–Weight Protocols for Authentication and Key Distribution. *IEEE/ACM Transc. on Networking*, 3(1), February 1995.

[BR93] M. Bellare and P. Rogaway. Entity Authentication and Key Distribution. In *Advances in Cryptography*, August 1993.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security Repelling the Willy Hacker.* Addison Wesley, 1994.

[CGHK94] P. Cheng, J.A. Garay, A. Herzberg, and H. Krawczyk. Modular Key Management Protocol. IETF draft–cheng–modular–ikmp-00.txt, November 1994.

[DBC92] *Great Circle Associates* D. Brent Chapman. Network (In)Security Through IP Packet Filtering. In *UNIX Security Symposium III Proceedings*, pages 63–76, 1992.

[DH76] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transc. on Information Theory*, IT–22(6), November 1976.

[IB94a] John Ioannidis and Matt Blaze. The Architecture and Implementation of Network–Layer Security under UNIX. In *Winter USENIX Conference Proceedings*, 1994.

[IB94b] John Ioannidis and Matt Blaze. *The swIPe IP Security Protocol.* IETF draft–ipsec–swipe–01.txt, June 1994.

[KN93] John Kohl and B. Clifford Neuman. The Kerberos Network Authentication Service (V5). Internet RFC 1510, September 1993.

[LJFK86] Samuel J. Leffler, William N. Joy, Robert S. Farby, and Michale J. Karel. Networking Implementation Notes, 4.3BSD Edition. In *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX–*

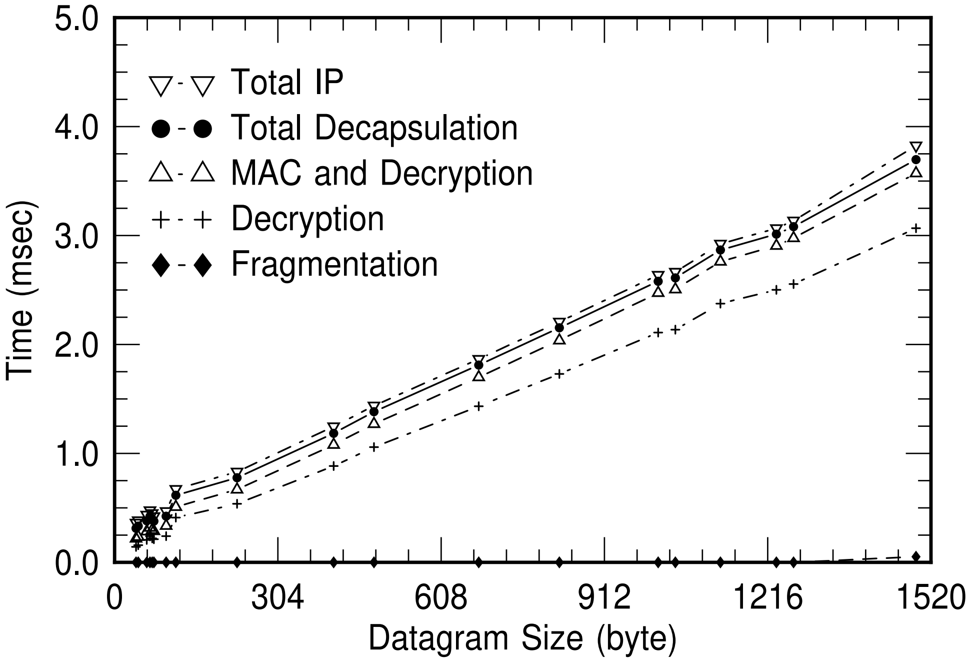

Figure 11: IP/IPST input processing time vs. datagram size on 50–MHz RS/6000 model 360

11 Edition. USENIX Association, April 1986.

[Pos81] J. Postel. Internet Protocol. Internet RFC 791, September 1981.

[Riv92] R. Rivest. Internet RFC 1321, April 1992.

[Tsu92] G. Tsudik. Message authentication with one-way hash functions. In *Proceedings of Infocom 92*, 1992.