The Architecture and Implementation of Network-Layer Security Under Unix

John Ioannidis
Columbia University
New York, NY 10027

ji@cs.columbia.edu

Matt Blaze
AT&T Bell Laboratories
Holmdel, NJ 07733

mab@research.att.com

ABSTRACT

swIPe is a network-layer security protocol for the IP protocol suite. This paper presents the architecture, design philosophy, and performance of an implementation of swIPe under several variants of Unix. swIPe provides authentication, integrity, and confidentiality of IP datagrams, and is completely compatible with the existing IP infrastructure. To maintain this compatibility, swIPe is implemented using an encapsulation protocol. Mechanism (the details of the protocol) is decoupled from policy (what and when to protect) and key management. swIPe under Unix is implemented using a virtual network interface. The parts of the implementation that process incoming and outgoing packets are entirely in the kernel; parameter setting and exception handling, however, are managed by user-level processes. The performance of swIPe on modern workstations is primarily limited only by the speed of the underlying authentication and encryption algorithms; the mechanism overhead is negligible in our prototype.

1. Introduction

Traditionally, system security has been addressed in an *ad-hoc* fashion and at a fairly high level, usually in the applications themselves. As applications become more distributed in nature and are relying more and more on the integrity of information received from their peers, it becomes attractive to consider common, general solutions to replace *ad-hoc*, application-specific security mechanisms. Most system-level security work has focused on the operating system and on arbitrating access to resources on a particular machine. Relatively little attention has been paid to securing communications in any consistent way. In modern distributed systems, however, which are characterized by a large number of single-user or single-purpose machines, the network itself is emerging as the best candidate for concentrating security efforts.

The explosively increasing size of the Internet, with its concomitant security problems, coupled with the wide range of services and applications that it supports, make IP-based networks a good choice for understanding and exploring network security issues. The rising mobility of users, the advent of wireless networking services, and the increasingly critical dependence of financial and commercial services on the Internet infrastructure, call for the adoption of authentication, integrity, and privacy features as *sine qua non* features of the network.

There have been a number of efforts aimed at providing security services at some layer in the network hierarchy [4,8,12,16]. However, no existing protocol is completely satisfactory for large-scale Internet-based deployment. Existing network-layer security protocols such as SP3 and NLSP implicitly impose a specific model of trust and policy; this manifests itself in the form of unwieldy features and a multiplicity of options. This unnecessary complexity frustrates efficient implementations and confuses mechanism with policy. Non-network-layer protocols, such as ADP and Kerberos, make it difficult to enforce security policies in a consistent manner, and often make it impossible to exploit their features without rewriting applications. Furthermore, non-network-layer solutions cannot be used to transparently enforce intermediate-hop security in an internetwork. swIPe's power lies in its minimalist structure; it avoids facilities that could best be provided elsewhere. Regardless, the techniques developed for swIPe can be readily adopted by implementations of any network-layer security protocol.

swIPe was designed to complement IP functionality by adding the necessary security features without altering the structure of IP itself. This is accomplished by encapsulating IP datagrams within IP datagrams of a new IP Protocol type (**`IPPROTO_SWIPE`**); these datagrams carry the payload of the original IP datagram, enough header information to reconstruct them at the remote end, and any additional security information that may be needed.

This approach has a number of advantages. First and foremost, since datagrams are encapsulated within other IP datagrams, they can transit parts of the network that know nothing about swIPe. Since the transport and higher-layer protocols never see swIPe datagrams, they need no changes and can employ the existing network infrastructure and interfaces. Since encapsulation and decapsulation can take place at any place where IP datagrams are processed (*i.e.*, either hosts or routers), swIPe may be used to implement both end-to-end and intermediate-hop security. Furthermore, a wide variety of security policies can be implemented simply by controlling when to encapsulate *outgoing* datagrams and when to accept *incoming* datagrams (which may or may not be swIPe-encapsulated). Finally, as our prototype experience demonstrates, swIPe can be implemented efficiently on a variety of modern Unix systems, such as SunOS, Mach, and BSDI.

swIPe provides three fundamental security services:

- Data confidentiality: protecting against unauthorized access to data being transmitted.
- Data integrity: protecting against alteration or future replay of traffic.
- Source authentication: network addresses are authenticated as part of the protocol.

It is important to point out that swIPe's protections are limited to the context of the network. Data not on the network itself, such as files, window contents, etc., may still require other protection mechanisms (*e.g.*, CFS [2]).

2. swIPe Protocol Overview

A complete specification of the swIPe protocol is beyond the scope of this paper. The reader is referred to [6] for the authoritative protocol description. This section gives a brief overview of the important features in swIPe.

Briefly, a swIPe system consists of three conceptual entities on top of the ordinary IP mechanisms: the *security processing engine*, the *key management engine*, and the *policy engine*. This three-way split is entirely conceptual; an implementation need not maintain strict barriers between them.

The policy engine is responsible for three tasks: examining outgoing packets to determine whether they require swIPe processing, examining incoming packets to determine whether they are to be accepted, and deciding the exact nature of the processing. Note that the policy engine implements the local security policy for the host, which must be coordinated with the security of its correspondents.

The key management engine establishes the session cryptographic variables used by the security processing engine. It also communicates with key management engines on other hosts to establish key associations, and is responsible for managing any required secure key exchanges. Cryptographic variables can be transmitted with pre-arranged secret keys, with an authenticated Diffie-Hellman exchange, or through some other user-defined mechanism.

The security engine applies the actual authentication, integrity, and confidentiality processing on individual datagrams, as controlled by the policy engine, and using keys provided by the key management engine.

It is useful to think of swIPe as an input layer that processes datagrams between the data link layer and IP, and as an output layer that pre-processes datagrams just before they reach IP from higher layer protocols (see Figure 1). Actually, packets being routed – passed from the input side to the output side of IP –must also pass through the output swIPe layer.

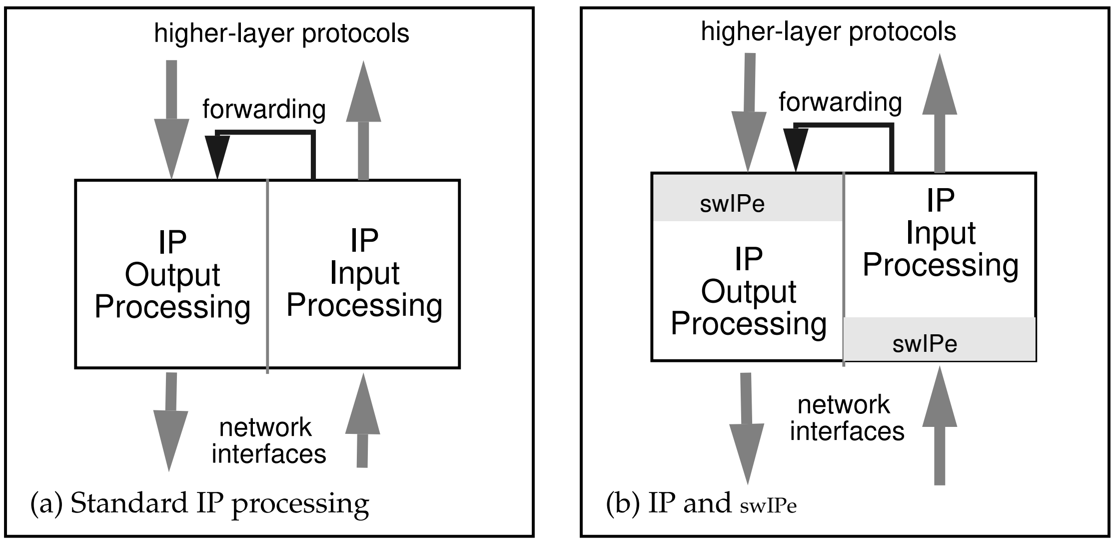

Figure 1: IP Processing with and without swIPe

IP datagrams are encapsulated within swIPe packets using an extension of IPIP, the IP-inside-IP protocol (used by Mobile*IP, as defined in [5,7]). A swIPe packet is itself an IP datagram, containing an IP header with the IP protocol field set to 94. The payload of this packet consists of the swIPe header, which in turn contains control information to direct the receiving side on how to process the datagram, any necessary security information (such as an authentication value and sequence number (to protect against replay)), and the original IP datagram, possibly encrypted. The packet format is shown in Figure 2.

Conceptually, swIPe output processing proceeds as follows. The policy engine examines the IP packet (in the simple case, only its destination address), and determines whether the packet requires encryption, authentication, or both. If not, the packet is simply sent down to IP. Otherwise, a swIPe header is generated; the security processing engine obtains the keys from the key management engine, applies the appropriate authentication and encryption algorithms, and the resulting encapsulated packet is sent to IP for delivery. Note that it is possible that the key management engine may be unable to supply a key immediately (*e.g.*, a secure key exchange protocol may first have to be initiated with a remote host). In these cases, the implementation may either re-queue the packet until all keys are available, or simply drop it and rely on higher-layer protocols to retransmit it later.

Input processing proceeds in roughly the opposite manner. When an IP datagram is received, the swIPe policy engine examines it; if the packet is already a swIPe packet, it is passed to the security engine, which processes the packet in a manner analogous to the output processing described in the previous paragraph.

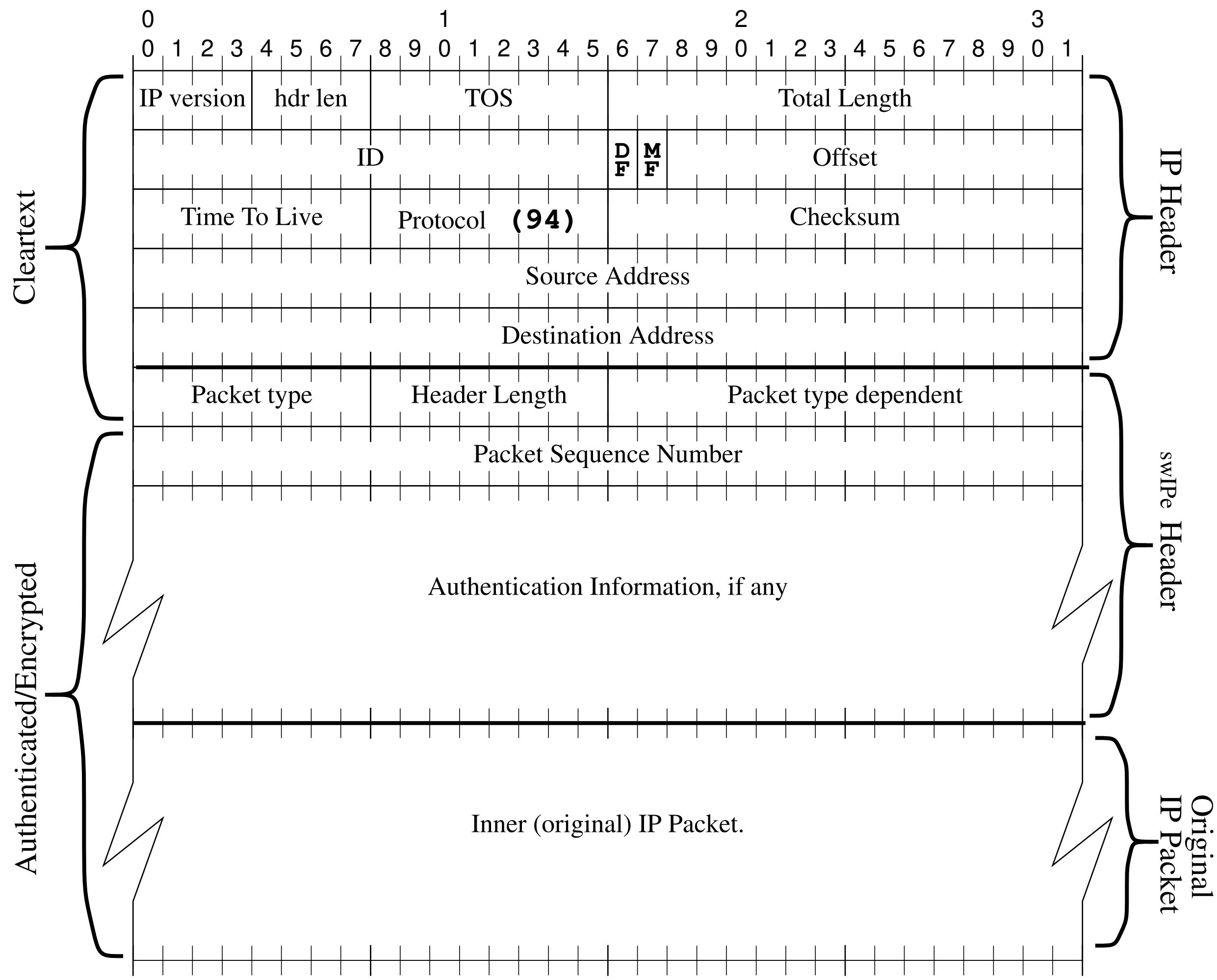

Figure 2: swIPe Packet Format

Otherwise, the policy engine determines whether the packet is admissible without authentication/encryption, and if so, simply passes it on to IP for regular input processing. If the packet is not admissible, it initiates exception processing, which could result in either a new key exchange, or simply an error message being recorded in the system log.

Note that the specifics of cryptographic variable exchange are not part of the swIPe protocol *per se*. swIPe provides an out-of-band mechanism for communication between key management agents through which necessary key exchanges take place.

Note also that swIPe does not require the use of any particular encryption or authentication scheme. Any encryption and cryptographic authentication functions can be used. Our implementation, described in the next section, supports DES [11] for encryption, and MD5 [14] seeded with a key for authentication. However, there is no reason why one could not use, *e.g.*, triple-DES, IDEA [9], or even Skipjack [10] for encryption, and SHS [13] for authentication, if desired.

swIPe can be used in a variety of configurations, each implementing a different security model. The simplest configuration is end-to-end security, as shown in Figure 3. Here, the individual hosts, **`alice`** and **`bob`**, use swIPe directly to communicate securely with each other. Both hosts must be equipped for swIPe processing.

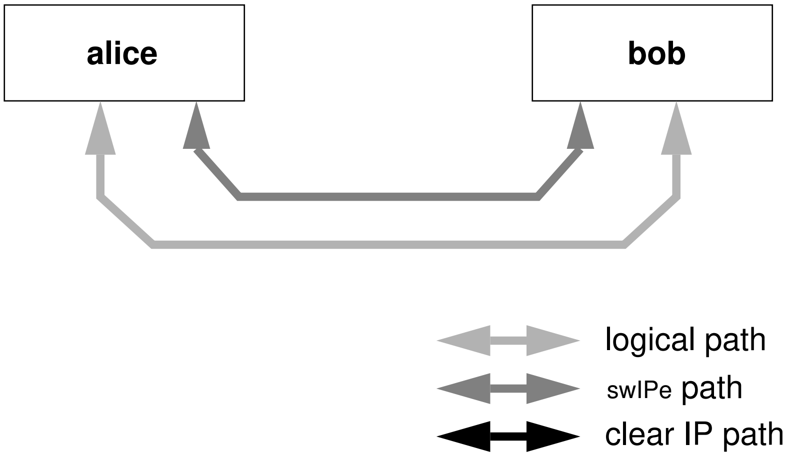

Figure 3: End-to-end swIPe operation

Other configurations are also possible in which the endpoints of communication are not swIPe-equipped. These configurations exploit the ability to run swIPe on routers as well as end hosts, and is the reason for encapsulating entire datagrams and not just the payload. Figure 4a shows the common network firewall as implemented under swIPe. The machine outside the firewall, **`alice`**, needs to communicate with the machine inside the firewall, **`bob`**. Since the firewall will not allow direct, unauthenticated IP connectivity, **`alice`** uses swIPe to deliver datagrams to **`frank`** (the firewall), which in turn decapsulates them and delivers them to **`bob`**. It is important to note that no machine inside the firewall (including **`bob`**) needs to run swIPe in order to securely communicate with outside hosts. This mode of operation can be extended to securely connect two or more networks over an untrusted link. Figure 4b shows **`alice`** and **`bob`** *tunneling* their traffic through a pair of swIPe firewalls, **`tina`** and **`tom`**.

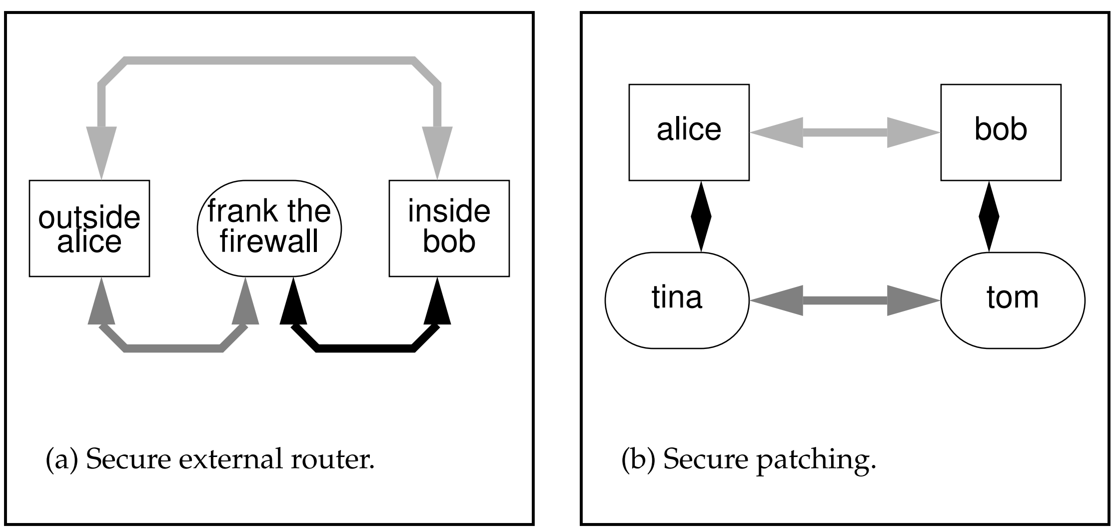

Figure 4: Alternate swIPe configurations.

3. Implementation under 4.3 BSD.

In order to investigate the feasibility of network-layer security in practical environments and to provide a testbed for experimenting with various security polices, we built a prototype implementation of swIPe. Our prototype runs on several BSD-derived platforms, including SunOS 4.1.2 on Sun SparcStation-2s, and CMU's Mach 2.5 running on 486-class PCs. This section describes our implementation under BSD-Tahoe-derived Unix systems, in particular SunOS.

While our description of swIPe involved three conceptual entities (the security, policy, and key management engines), our implementation blurs the boundaries for performance and practicality reasons.

Part of the prototype is implemented within the Unix kernel (in particular, the security and policy engines, and the caching of session keys), and part is implemented at user level (in particular, most of the key management functions, as well as the policy configuration interface).

We have adopted a simple model of security policy, in which each host determines on a host-by-host basis the precise security transformations, acceptance policies, and keys to use. Future security policies should allow more sophisticated policy specifications, in which both coarser (groups of hosts) and finer (per-protocol or per-application) control is possible.

To simplify the implementation, we isolate all complex policy operations within easily modified user-level code. At the same time, performance requirements preclude consulting a user-level process for each packet. To balance these competing constraints, swIPe's operational parameters are specified through user-level interfaces to policy and key tables that are maintained in kernel space. Most packets can be processed entirely within the kernel simply by consulting those tables; only packets that cannot be processed according to existing table entries cause upcalls to user-level processing. swIPe adds four tables to the kernel. Two tables, the *input* and *output key tables*, contain the current session keys for a particular remote host; they are indexed by the address of the encapsulating packet. The session key entries also have a "time to live" which is decremented each time they are used. The other two kernel tables, the *policy tables*, describe how encapsulated and non-swIPe packets should be handled; they are indexed by the host IP address of the encapsulated (non-swIPe) packet. The input policy table describes the kind (if any) of swIPe processing (encryption, authentication, or both) that is expected of packets from a particular source IP address, and the address allowed to encapsulate packets for it. The output policy table contains the converse data, describing the kind of processing to perform for a given destination address and the address to which the encapsulated packets should be sent for it. Having two sets of tables, one for keys and one for policy, indexed separately, is the mechanism through which non-end-to-end configurations as described in the previous section are implemented.

The swIPe implementation includes kernel modifications for processing incoming and outgoing IP datagrams and processing policy table management **`ioctl()`** calls, a user-level *key management daemon*, a user-level *policy daemon*, and a set of tools for configuring swIPe policy (by issuing the appropriate table-management **`ioctl()`** calls).

The swIPe kernel code for processing outgoing datagrams is contained within a virtual network interface (called **`sw`***n*, where *n* is **`0`**, **`1`**, etc.). This virtual interface appears to the rest of the system as if it were a conventional network interface; this technique is similar in spirit to the one described in [1]. Recall that conceptually, the policy engine filters all outgoing IP datagrams to determine whether the packet requires processing by swIPe. Since in our prototype only the destination IP address is used to select these packets, we use the existing BSD routing mechanism as our primary outgoing policy filter. At configuration time, a route is set to **`sw0`** for all hosts and networks with which swIPe processing is desired. It is also necessary to specify the kind of processing (encryption, authentication, or both) desired for each host and where the encapsulated packets should be sent. This is done through a special **`ioctl()`** call that creates an entry of the appropriate kind in the outgoing policy table. Another **`ioctl()`** sets up session key entries in the key table (with a time-to-live of **0**) for the encapsulating address. These **ioctl()**s are generally called by the key and policy daemons according to user-level configuration files.

When an outgoing packet is routed through **sw0**, swIPe first looks up the destination IP address in the output policy table to find the address to which the encapsulated packet should be sent. If no entry exists, the packet is sent, via the upcall mechanism (described below) to the policy management daemon. (Note that when no entry exists for an outgoing packet that was routed to sw0, it is because the destination host's address is on a network for which swIPe processing is required but to which no packets have previously been sent). If the lookup was successful, swIPe then looks up the encapsulating address in the output key table to find the current session keys for the appropriate operations. If this lookup fails, or if the keys are expired, the packet is sent via the upcall mechanism to the key management daemon. The packets that caused this exception processing are not delivered to their corresponding higher-layer protocol; their retransmission once a key is established is handled entirely by the higher-layer protocol.

Most outgoing packets have valid table entries and do not cause exceptions. These packets are encrypted and/or authenticated according to the policy rules and keys in the tables, and then encapsulated as the payload of a swIPe packet. The resulting packet is then sent to its destination (which may not be the same as the original packet) via the appropriate interface according to the standard routing tables (ignoring any **sw0** entries). The processing of outgoing packets in the kernel is summarized in Figure 5. Note that packets from the key management daemon bypass swIPe processing, since this traffic must usually complete as a prerequisite to passing any swIPe-protected data.

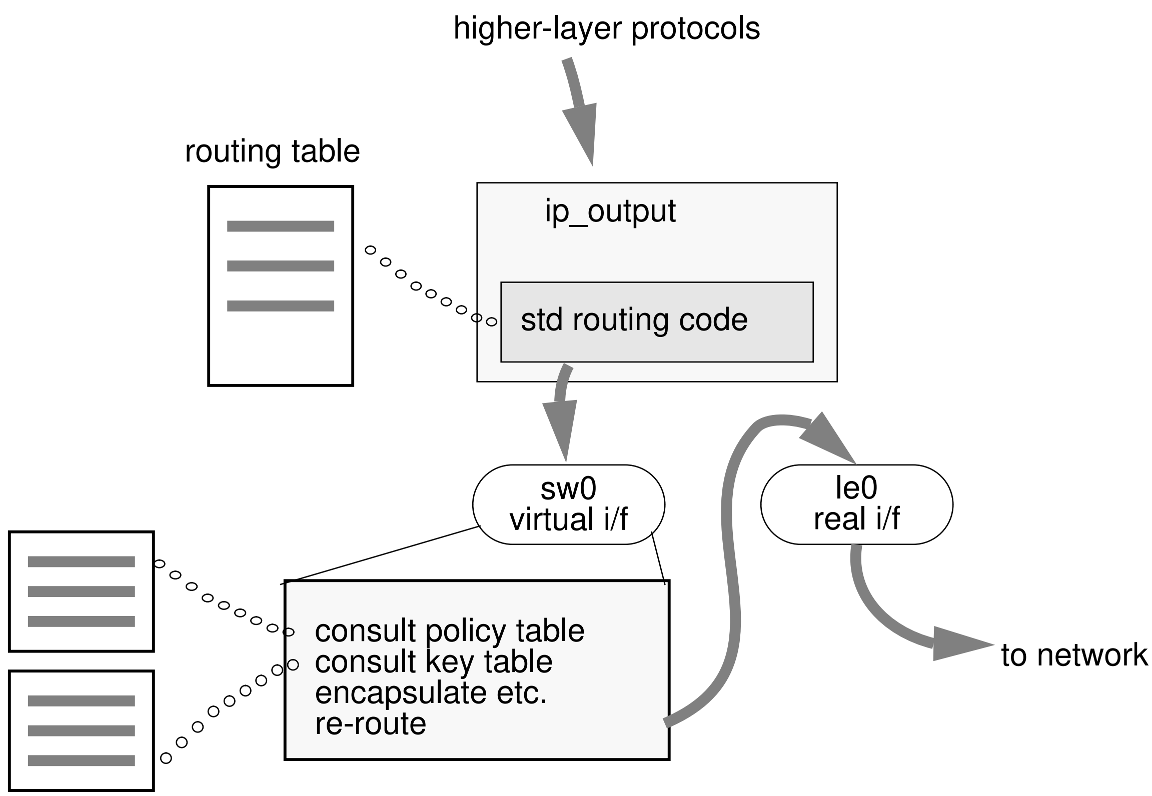

Figure 5: swIPe output processing.

Because no existing mechanism in 4.3BSD demultiplexes incoming packets by remote address (as the routing mechanism does for outgoing packets), input processing is more closely coupled to the rest of the kernel network code. swIPe modifies the standard kernel IP input routine to add a rudimentary packet filtering mechanism. The filter examines all incoming packets, whether they are swIPe-encapsulated or not, to determine if they should be accepted and passed on to the appropriate higher-layer protocol handler. The filter is controlled by the input policy table indexed by remote (source) IP host (or network) address, similar in structure to the output policy table.

Observe that there are several kinds of incoming packets that the filter must handle. First, consider non-swIPe packets that arrive. If they arrive from a host from which swIPe packets were expected (according to the table), they are dropped and a message is sent to the system log. Packets for which the table specifies that swIPe processing was not expected are passed on to the higher-layer protocol or routing mechanism. Unencapsulated packets that arrive from a host for which no table entry exists cause an exception and are sent to the policy daemon, which examines them and creates an appropriate table entry. Packets sent to the daemon are otherwise dropped. Note that packets intended for the key management engine (with protocol **`IPPROTO_swIPe`**) bypass these checks and are delivered directly.

Encapsulated swIPe packets are first decapsulated and decrypted/authenticated according to the keys in the input key table as indexed by the address of the encapsulating packet. If no key entry exists, or if the keys do not decrypt and authenticate properly, the packet is sent up to the key management daemon, which initiates a session key request with the remote host. Otherwise, the decapsulated packet's source address is looked up in the policy table to verify that its encapsulating packet came from the proper address and that the correct kind of processing (encryption, authentication) was applied. If everything is acceptable, the packet is delivered to the appropriate higher-layer protocol or routing mechanism. Otherwise, a console message is printed and the packet is dropped. If no input policy entry exists for the encapsulated source address, the packet is sent to the policy daemon and dropped.

Conceptually, the kernel policy and key tables can be thought of as cache entries for a database maintained by the policy and key daemons. Since the number of hosts with which a machine might exchange traffic is potentially very large, it is not practical to maintain table entries in the kernel for every potential correspondent. Instead, swIPe configuration is specified in user-level files read by the appropriate daemons. When a packet arrives at the policy daemon, the destination address is looked up in the appropriate file; if a match is found, an **`ioctl()`** is issued to configure the policy in the kernel. This allows policy to be specified concisely for large classes of addresses; while the kernel table entries are on a host-by-host basis, the daemon's tables can look up policy by their network or subnet address as well.

Key distribution uses either public or secret key techniques. In the public key version, each host's key management daemon maintains a list of RSA [15] public signature keys for each of the other swIPe hosts. When a session key is required, a Diffie-Hellman key exchange is initiated, with the components of the transaction signed with each host's private key. Public key distribution is static and manual; this is obviously an area for much future research. This aspect of our prototype is primarily useful as a proof of concept. In the secret key version, each host maintains a unique, static key for each other host which is used to encrypt the session keys. Again, the kernel table can be thought of as a cache of session keys, with the definitive keys managed by the daemon.

Our implementation of swIPe can be used in a variety of ways: in its simplest form, it provides host-to-host authentication and encryption. It can also be used to implement all the 'firewall' configurations described in the previous section.

The basic swIPe implementation adds less that 3000 lines of code to the kernel; the actual DES and MD5 code adds approximately another 3000 lines.

4. Performance

Our swIPe prototype is implemented entirely in software; this represents a somewhat unconventional approach to cryptographic protection of network traffic. Algorithms such as DES have traditionally been thought too slow to permit their use in high-bandwidth applications without adjunct hardware. However, several key factors make software implementations of network encryption increasingly practical: modern workstations are becoming sufficiently powerful to execute cryptographic operations at speeds equal to previous-generation cryptographic processors; network bandwidth is not increasing at the same rate as processor speed, and finally, distributed applications rarely use even a small fraction of available network bandwidth. While our experience with swIPe reinforces our belief that encryption is becoming practical in software, swIPe itself does not depend on any particular software or hardware architecture.

We instrumented the SunOS 4.1.2 kernel to measure the costs of the various components of our implementation. All measurements were taken by using the internal SparcStation microsecond-resolution clock. The clock was read in key locations of the code being timed, and the resulting timestamps and time intervals were deposited in a kernel buffer. A user-level process would periodically empty the buffer and record the results. The measurement error was within approximately 4 microseconds.

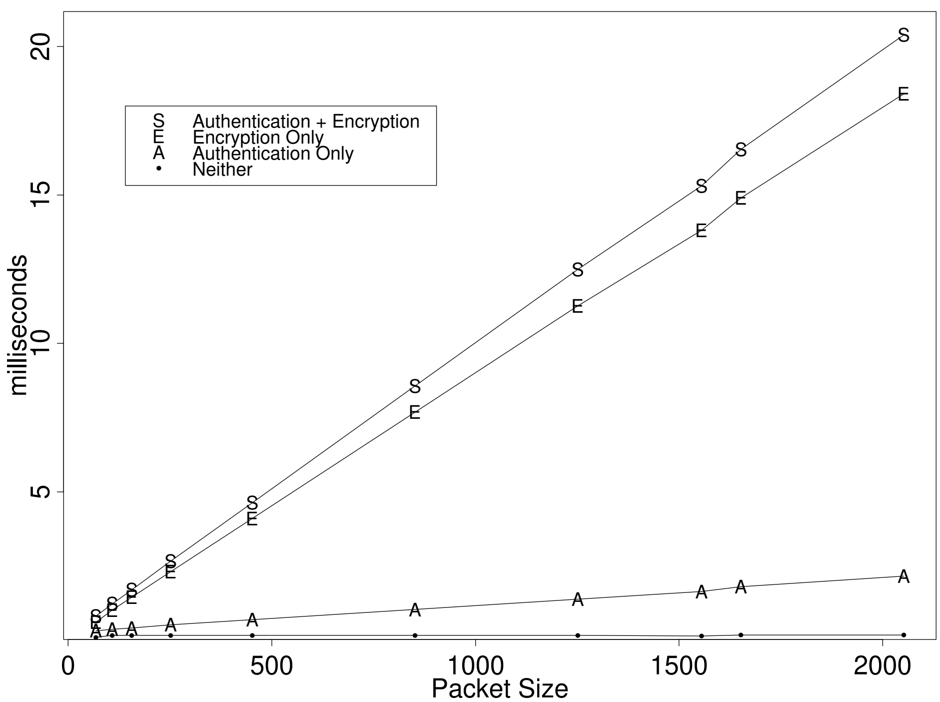

Figure 6: swIPe processing overhead

The overhead of the swIPe protocol processing itself is minimal, adding approximately 100 microseconds to the total packet processing time regardless of packet size. For small datagrams, this is about half the time it takes the kernel to deliver them to the hardware interface, diminishing further in significance as the datagram size increases. This overhead is dwarfed by the amount of time it takes to compute a cryptographic hash function, which in turn is dwarfed by the time it takes to perform a single pass DES in software. On a SparcStation-2 class machine, processing a 100-octet datagram adds roughly 200 microseconds for computing an MD5 cryptographic checksum, and another 900 microseconds for encrypting using a highly-optimized software implementation of DES. Processing a 1000-octet packet requires roughly 1000 microseconds for the MD5 checksum and 10,000 microseconds for the DES encryption. Figure 6 shows the total time required to process a swIPe packet in the kernel as a function of the packet size. The four curves show the processing time the four possible processing combinations: no protection, only authentication (MD5), only encryption (DES), and both authentication and encryption.

The timing ICMP ECHO packets is consistent with these numbers. Namely, a 56-octet ping gives a round-trip delay of 8 milliseconds, while a 1000-octet ping gives a round-trip delay of 47milliseconds. These numbers include four encryption/decryptions and four authentication operations (when the packet is sent, received, returned, and received again). TCP throughput between two SparcStation-2s on a lightly loaded Ethernet is approximately 440kbps when using DES encryption; the observed variance (~30kbps) obliterates any fine-grained measurement of the added effect of authentication. When using MD5 authentication only, we observed TCP throughput at approximately 3400kbps. For comparison, the TCP throughput of the unmodified machines (without swIPe) is roughly 6000kbps.

It is important to put these numbers in perspective. Interactive traffic consists mostly of small packets; hence, adding authentication and encryption between communicating hosts adds a fixed delay of 2-3 milliseconds, which is not perceptible considering that this is usually the delay associated with going through a single router. Large packets are usually associated with batch transfers or remote filesystem operations, and are usually already bound by disk I/O rates rather than network rates. Note that on a SparcStation-2, swIPe can process datagrams at roughly one megabit per second, which, although not transparent, is at least within an order of magnitude of the bandwidth of a typical NFS server. Of course, hardware-assisted cryptographic functions can further reduce the effects of this bottleneck, as could the use of lower latency cryptographic algorithms (e.g., DES's OFB mode, which can use idle processor cycles to precompute cryptographic masks).

5. Conclusions

In developing a network security protocol, it was important to keep both interoperability with existing systems and a migration path in mind. To this end, swIPe functionality does not depend on any IP-specific features, and can therefore be easily adapted to other connectionless network protocols.

swIPe is not only a fully-functional network security protocol in itself, but also provides the framework for experimenting with alternative security algorithms and policies, and, more important, large-scale key management strategies. It is perhaps in these areas that we will see much of the interesting research in network security systems. It is our hope that researchers and developers of network and other large-scale security systems will adopt swIPe as a vehicle; while the current implementation is by no means stable enough for general distribution, interested parties are encouraged to discuss their plans with the authors.

Acknowledgments

This work owes its conception to many fruitful discussions, guidance, and encouragement by Phil Karn. The implementation of the encryption code was aided greatly by discussions with Jack Lacy. The implementation and testing were carried out over many late evenings at the Mobile Computing Lab at Columbia University.

References

[1] S. M. Bellovin. Pseudo-Network Drivers and Virtual Networks – Extended Abstract. In *Usenix Conference Proceedings*, pages 229–244. Usenix, January 1990.

[2] Matt Blaze. A cryptographic file system for Unix. In *First ACM Conference on Communications and Computing Security*, Fairfax, VA, November 1993.

[3] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory}*, IT-22:644–654, 1976.

[4] D. P. Anderson *et al.* A Protocol for Secure Communication in Large Distributed Systems. Technical Report UCB/CSD 87/342, University of California, Berkeley, February 1987.

[5] John Ioannidis. *Protocols for Mobile Internetworking*. Ph.D. thesis, Columbia University in the City of New York, 1993.

[6] John Ioannidis, Matt Blaze, and Phil Karn. swIPe: The IP Security Protocol. *To appear.*

[7] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire Jr. IP-Based Protocols for Mobile Internetworking. In *Proceedings of SIGCOMM'91*, pages 235–245. ACM, September 1991.

[8] ISO/IEC JTC1/SC6. ISO-IEC DIS 11577 – Information Technology –Telecommunications and Information Exchange Between Systems – Network Layer Security Protocol, November 29 1992.

[9] X. Lai and J. Massey. A Proposal for a New Block Encryption Standard. *Proc. EUROCRYPT 90*, pages 389–404, 1990.

[10] Mykotronx, Inc. MYK-78 Encryption/Decryption Device. Data Sheet, 1993.

[11] National Bureau of Standards. *FIPS Publication 46-1: Data Encryption Standard*, January 1988.

[12] National Institute of Standards and Technology. *NISTIR 90-4250: Secure Data Network System (SDNS) Network, Transport, and Message Security Protocols*, February 1990.

[13] National Institute of Standards and Technology. *Publication YY: Announcement and Specifications for a Secure Hash Standard (SHS)*, January 22, 1992.

[14] R.L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. Internet Activities Board, April 1992.

[15] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[16] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An authentication service for open network systems. *Proc. Usenix Winter Conference*, February 1988.