The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Implementation of IPv6 in 4.4 BSD

Randall J. Atkinson, Daniel L. McDonald, Bao G. Phan,
Craig W. Metz, and Kenneth C. Chin
Information Technology Division, Naval Research Laboratory

Implementation of IPv6 in 4.4 BSD

Randall J. Atkinson, Daniel L. McDonald, Bao G. Phan, Craig W. Metz*, & Kenneth C. Chin

Information Technology Division, Naval Research Laboratory

Abstract

The widespread availability of the TCP/IP protocols in early versions of BSD UNIX fostered the currently widespread use of those protocols in commercial products. Rapid depletion of the IPv4 address space has caused the Internet Engineering Task Force to design version 6 of the Internet Protocol (IPv6). IPv6 has some similiarities with IPv4, but it also has many differences, most notably in address size. This paper describes our experience creating a freely distributable implementation of IPv6 inside 4.4 BSD, with focus on the areas that have changed between the IPv4 and IPv6 implementations.

1 Introduction

During the past decade, the worldwide Internet has grown at exponential rates, not only in North America but also in Europe and Asia. [Lot92] This, combined with suboptimal address allocation practices, has led to increasing depletion of the IP version 4 (IPv4) address space. One direct result of the IPv4 address depletion was that the Internet Engineering Task Force (IETF), began working to create a revised version of the Internet Protocol (IP). This effort is called Next-Generation IP (IPng). The resulting protocol is IP version 6 (IPv6). When the IPng effort began, there were several contenders, but in July 1994 the SIPP proposal became the primary basis for IPv6.

The widespread availability of TCP/IPv4 in early versions of BSD UNIX was crucial to the success and deployment of the Internet technologies. In order to help make Next-Generation IP as widely available, the authors began working with the Simple Internet Protocol (SIP) Working Group of the IETF in 1992.[Dee93] As SIP evolved into SIPP [Hin94] and then into IPv6, the authors began prototyping, initially in BSD Net/2 and currently in 4.4 BSD. Our primary development systems were Sun SPARC workstations and i486 systems running 4.4 BSD [1].

Implementation issues, rather than the details of the IPv6 protocol, are the focus of this paper. A number of implementation issues arose with IPv6 and have been resolved. Obvious issues, such as supporting 128 bit addresses instead of 32 bit addresses, are discussed in addition to the less obvious issues of how to implement IPv6 security inside a BSD kernel. We assume that the reader is somewhat familiar with the IPv6 protocol [DH95] and the 4.4 BSD-Lite implementation of IPv4. Figure 1 shows a rough overview of 4.4 BSD-Lite's Internet implementation, along with some of the new modules for IPv6. To add a new version of IP, many of the surrounding modules had to be modified as well.

2 Changes in Basic IP Functions

2.1 Differences in packet format

Perhaps the most obvious difference between IPv6 and its predecessor is the packet format. Although some in the Internet community felt that 64 bit addresses were sufficiently large, others insisted that 128-bit addresses were needed so that plug-and-play address assignment similar to ISO ES-IS could be supported. Many of the IPv4 header fields that were unused in practice (Figure 2) were eliminated or moved to options, making the IPv6 base header (Figure 3) more streamlined. One significant addition to the header is the *Flow Identifier* which is an important hook for resource reservation techniques [ZBE+93] currently being developed within the IETF.

The sparse IPv6 header is optimized for minimal processing. An IPv6 router needs only to verify the version number, inspect the destination address,

*Although Craig W. Metz is with Kaman Sciences Corporation, he may be reached at NRL.

[1] The systems running 4.4 BSD (encumbered) have had the 4.4 BSD-Lite networking changes incorporated into them. Some call this a BSD Net/3 system.

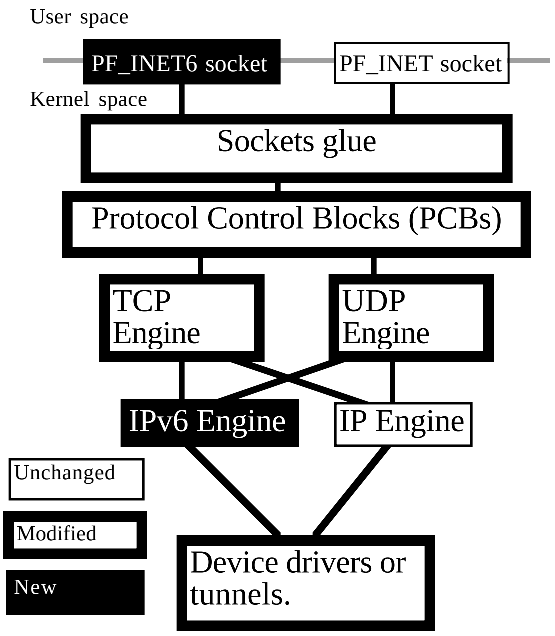

Figure 1: Simple Overview of 4.4 BSD-Lite Internet Modules

decrement the hop counter, and process hop-by-hop options if they are present. (The flow label can be used to optimize this process further.) An IPv4 router has to perform everything an IPv6 router does, as well as verify and recompute the header checksum, and fragment the datagram further if needed. An IPv6 destination host initially only has to check the validity of the version and destination address. If there are options, they are daisy-chained and indicated by the Next Header field. Otherwise, a higher-level protocol (e.g. TCP) is the next header processed. An IPv4 destination host has to verify not only the version and destination address, but the IP header checksum as well.

2.2 Protocol Processing

A number of the more recently developed IPv4 optional features are mandatory in IPv6. Other features, such as cryptographic security, are new with IPv6[2]. These have caused a number of changes in IP protocol processing.

IPv6 daisy-chains optional headers after the base header. Our implementation pre-parses an IP packet into its constituent headers and upper-layer protocol data as part of the initial IPv6 input processing. Although this does degrade performance, it has simplified the processing of optional IPv6 headers. We plan to create a *fast path* around the preparsing code for packets containing no optional headers.

The *Path MTU Discovery* [MD90] technique for avoiding IP fragmentation in routers is mandatory for IPv6. IPv6 does not have any intermediate fragmentation and instead relies on Path MTU Discovery and end-to-end fragmentation. Our implementation stores Path MTU information in host routes. Host routes are automatically created for IP communications originating on the local machine. Storing this information in the routing table makes this data available to TCP, UDP, and ICMP. IPv6 requires a minimum MTU of 576 bytes, which is much larger than the 68 byte minimum MTU of IPv4. However, even this larger size might be too small if certain IPv6 options, such as the Hop-by-Hop Options Header (which can be up to 2048 octets), are used. In such cases, end-to-end fragmentation will be required.

3 Security Processing

Two cryptographic security mechanisms have been defined for IPv6 [Atk95c]. One, known as the Authentication Header (AH), provides authentication without confidentiality[Atk95a]. The second, known as the Encapsulating Security Payload (ESP), provides confidentiality through encryption of packet contents. [Atk95b] ESP has two modes. The first mode, known as *Transport-mode*, encrypts only the upper-layer header and data (such as TCP, UDP, or ICMP) and leaves the IP header in the clear. The second mode, known as *Tunnel-mode*, encrypts an entire IP datagram, prepending an additional cleartext IP header outside the encrypted IP datagram so that the packet can be routed. The implementation of these mechanisms broke new ground within the BSD kernel. In addition to implementing the Authentication Header and both modes of ESP, we also implemented the kernel support required to manage network security associations, including the cryptographic keys.

The IPv6 security mechanisms can use any appropriate encryption or authentication algorithm. The mandatory algorithms for a compliant implementation are keyed MD5[MKS95b] for authentication, and DES-CBC[MKS95a] for encryption. Both algorithms are in this implementation. To implement a new ESP or AH algorithm, the kernel must be recompiled with support for the new algorithms in place. Other algorithms, such as triple-DES, are be-

[2] The cryptographic security recently standardised for IPv4 and IPv6 was originally designed for use with IPv6 and later adapted for use with IPv4.

<table>
<tr><td>Version</td><td>Hdr Len</td><td colspan="2">Type of Service</td><td colspan="4">Packet Length</td></tr>
<tr><td colspan="4">Fragment Identification</td><td colspan="4">Flags and Fragment Offset</td></tr>
<tr><td colspan="2">Time-To-Live</td><td colspan="2">Protocol</td><td colspan="4">Header Checksum</td></tr>
<tr><td colspan="8">Source Address</td></tr>
<tr><td colspan="8">Destination Address</td></tr>
</table>

Figure 2: IPv4 Packet Format

<table>
<tr><td>Version</td><td>Priority</td><td colspan="6">Flow Label</td></tr>
<tr><td colspan="4">Payload Length</td><td colspan="2">Next Header</td><td colspan="2">Hop Limit</td></tr>
<tr><td colspan="8">Source Address</td></tr>
<tr><td colspan="8">Destination Address</td></tr>
</table>

Figure 3: IPv6 Packet Format

ing implemented by others. Later in this paper, we discuss why it is straightforward to add support for additional cryptographic algorithms.

Both ESP Transport-mode encryption and Authentication Header output processing are normally performed immediately before any fragmentation on outgoing packets and after reassembly on the input side. They are done this way because, except for fragmentation, they need to operate on the packet as it will appear on the wire. For example, the source address for the packet from a multi-homed system must be known before encryption or authentication can take place.

3.1 Security Associations

A fundamental concept behind IP security is the *Security Association*. A Security Association contains all of the configuration data for a particular secure session between two or more systems communicating via IP. For example, the security services in use (AH or ESP), the cryptographic algorithm(s) in use, the cryptographic key(s) in use, the key lifetimes, the Security Parameters Index (SPI), and the sensitivity level (e.g. Unclassified, Secret) of the session are all components of a Security Association. In order to support multicast as well as unicast, all Security Associations are one-way from source to destination. So a typical telnet session would need two Security Associations, one in each direction.

Security associations are stored in a table inside the kernel. A module called the *Key Engine* controls access to the table. The Key Engine allows kernel services, such as the IPv6 module, to obtain security associations for inbound and outbound packets. The Key Engine also communicates with user-level key management programs so that key management may be implemented properly. The relationship between the key engine and user-level key management programs is similar to the relationship between the routing socket[Skl91] and programs such as *gated(8)*.

3.2 Security Processing Structure

The authentication processing function is split into three major parts. The first, a keyed message digest function, is selected on a per–association basis through an *algorithm switch* that calls the appropriate computation function. The second, the header processing routines, finds the appropriate security association and policy actions for the packet and either builds or parses the actual option header for authentication. The third part is the meat of the authentication function. This routine walks the packet, header by header, zeroing header fields that vary unpredictably end-to-end, and passing other header fields and the packet data into the keyed message digest function. The resulting message digest data can be either inserted into the outgoing header or, in the case of an incoming packet, checked with the one in the header. The keyed message digest functions are treated in the AH calculation function as stream operations; any necessary blocking and padding must be handled by the implementation of the keyed message digest functions.

The encryption processing function is split into similar parts. The first, an encryption/decryption function, and the second, a transform header construction and parsing function, are selected on a per–association basis through an algorithm switch. Because almost all of the header format can vary depending on which cryptographic transform is being used, it is necessary that both the cryptographic functions and the header processing functions be switchable. There is a generic reblocking function that runs a specified encryption or decryption function over the data while arranging it into properly sized blocks. Block-oriented encryption and decryption functions require the encrypted data to be an integral number of cryptographic blocks.

3.3 Output Security Processing

Immediately before IP fragmentation is performed, `ipv6_output()` calls an IP security output policy function, `ipsec_output_policy()`, to determine whether this packet needs security. This function examines the system security level configured by the administrator and the socket security level requested by the process on the socket. The function is able to examine the socket security level because each outgoing packet data chain now contains a back pointer to the socket that sent the packet. The security output policy function then examines the system-wide security policy and the socket-requested security policy and applies the more paranoid of these policies to the outgoing packet.

The `ipsec_output_policy()` function is also responsible for making the `getassocbysocket()` call into the *Key Engine* to obtain *Security Association* data for the outgoing packet. If the Key Engine has the appropriate Security Associations, it provides access to them. If no appropriate Security Association exists and a key management daemon is running, then the Key Engine sends a *Request* message to that daemon and informs the output policy function that the Security Association has been delayed. If no appropriate Security Association exists and no key management daemon is running, then the Key Engine returns an error to `ipsec_output_policy()`. If this error occurs, it will eventually be presented to the user as the newly defined IP Security processing error, `EIPSEC`.

If IP security is needed and all appropriate security information is available for the outgoing packet, then the output security policy function will return both an indication of which services are needed and pointers to the appropriate Security Associations. The IP Output function then makes the appropriate calls to apply outgoing security services and then sends the packet out. If any errors occur during security output processing, the packet will be dropped and the user will be given the `EIPSEC` error mentioned above. In the future, we might enhance the `getassocbysocket()` call to provide the user identification or `uid` associated with the network socket so that the Key Engine can provide finer granularity of keying. The current implementation does support both shared (i.e. host-oriented) keys and also unique (i.e. socket-oriented) keys.

3.4 Input Security Processing

For incoming packets, the task is significantly easier. When an Authentication Header or Encapsulating Security Payload header is encountered, it is processed by calling the appropriate IP security input function (either `ipsec_ah_input()` or `ipsec_esp_input()`). That function reads the Security Parameters Index (SPI) contained in the cleartext portion of the received packet and makes a `getassocbyspi()` call into the Key Engine to obtain the correct Security Association for the received packet. If this call succeeds, the security input processing is performed and the appropriate security-related flag is set. The packet data chain has two new flags, both initially cleared on input, called `M_AUTHENTIC` and `M_DECRYPTED`. These flags indicate that the packet passed authentication processing and encryption processing, respectively. If any security input processing fails, the packet is dropped and appropriate kernel statistics counters are incremented. A modified *netstat(8)* is supplied that can display these statistics for the system administrator. If more than one form of security has been applied, then the packet will go through more than one security input processing function.

The input security processing code also performs special checks comparing the outer IP source address and the (previously encrypted) inner IP source address for the case when an IP datagram is tunnelled inside another IP datagram and either the Authentication Header or the Encapsulating Security Payload is present. These checks are intended to prevent an adversary system from encapsulating a forged packet inside an authenticated or encrypted legitimate packet and tricking the receiving system into believing the forged packet was authentic. If these source address checks fail, then the `M_AUTHENTIC` or `M_DECRYPTED` flags on the received packet data chain are cleared.

After security input processing is completed, the normal input processing resumes. Once the packet reaches the transport layer, the transport layer's input function, for example `tcp_input()`, calls `ipsec_input_policy()` to perform an input security policy check. The incoming packet is dropped if it does not meet the requirements for authentication or encryption that exist for its destination socket. Because `ipsec_input_policy()` checks not only the socket security requirements but also the system-wide security requirements, the system administrator can mandate a minimum security level for all normal network connections.

3.5 Policy Separation

The separation of the policy engine from the mechanisms allows per-socket security selections and administrative security selections to be combined in sophisticated ways. For instance, an administrator could require that packets coming in on a certain range of privileged ports must come from a privileged port and must be authentic in order to protect the administrator's system from potential abuses. The current policy engine only implements simple system-wide decisions (e.g., drop all non-authentic packets, always use authentication if we have a security association that will facilitate it) in conjunction with application requested socket security. Enhancements to the security policy engine are planned for the future.

3.6 Algorithm-independence

Care was taken to provide multiple levels of indirection to take advantage of the algorithm-independent nature of the Authentication Header and Encapsulating Security Payload (ESP) specifications. Both implementations use an algorithm switch, which is indexed by a value in the security association, to support multiple algorithms concurrently and allow easy addition of new message digest and encryption functions. This switch is more complex for ESP, because almost all of the ESP header format can change as a function of the transform in use. For this case, the switch allows implementors to specify the header processing code and the encryption code separately for greater flexibility. For instance, someone wanting to substitute the IDEA algorithm [LM91] for the default DES-CBC algorithm but still use the same basic header format could create a new algorithm switch entry that uses the same header processing functions as DES-CBC [MKS95a] but calls the IDEA encryption functions instead. Different algorithms will have different performance impacts. Supporting multiple algorithms in the kernel does not exact a significant performance penalty.

4 Changes to ICMP and IGMP

The Internet Control Message Protocol (ICMP) is perhaps not as widely known as TCP or UDP, but it performs a critical function in keeping the network operating smoothly. The Internet Group Membership Protocol (IGMP) is integral to IP multicasting. ICMP for IPv6 is sufficiently different that it is now sometimes referred to as ICMPv6 [Pos81][DC95].

Despite having similar header syntax, ICMPv6 differs from ICMP for IPv4 in four major ways. First, ICMPv6, like TCP and UDP, requires a pseudo-header to be included in its checksum calculation. Second, the difference between informational messages (e.g. *Echo*) and error messages (e.g. *Port Unreachable*) is now indicated by the high bit in the ICMPv6 message type. Third, ICMPv6 absorbs the functions of the formerly separate IGMP [Dee89], ARP [Plu82][FMMT84], Proxy ARP, and ICMP Router Discovery [Dee91] protocols. Finally, ICMPv6 also adds support for stateless address auto-configuration. Because ICMP is above the IP layer, all of these functions can now be authenticated and or encrypted using the IP security mechanisms, as long as appropriate security associations exist. Sites that wish to bootstrap securely can now do so.

4.1 Traditional ICMP and IGMP

ICMPv6 retains the functions traditionally performed by ICMP and IGMP. The *Echo* and *Echo-Reply* messages, utilized by ping(8), are still part of ICMPv6. Unreachability of varying forms is indicated by the ICMPv6 *Unreachable* message type. Extensions have been added to indicate unreachable on-link neighbors, as well as errors with strict source routing. A *Message Too Big* message indicates when an IPv6 datagram is too large for a link on its path. Path MTU discovery [MD90], a requirement for IPv6, is implemented using these messages. *Parameter Problem* messages indicate invalid IPv6 option fields, as they do in IPv4's ICMP. *Time Exceeded* messages indicate either a hop limit that has decremented to zero, or that an IPv6 reassembly has timed out.[3]

ICMPv6 has three additional informational messages: *Group Report*, *Group Query*, and *Group Terminate*. The first two behave just like the IGMP *Report* and *Query* messages. The *Group Terminate*

[3] This implementation cannot send Time Exceeded messages for IPv6 reassembly timeouts; the "offending packet" needed for the ICMPv6 message is no longer available for transmission because reassembly is occurring.

message is an optimization so that routers can be informed more quickly about hosts leaving multicast groups.

4.2 Address Auto-Configuration and Router Discovery

The Internet community mandated that IPv6 support simple address auto-configuration for hosts. IPv6 has two solutions to this problem. The first approach is to use an optional configuration protocol, such as DHCPv6. This solution is beyond the scope of this paper. The second approach, known as stateless address autoconfiguration, is required, and is implemented in ICMPv6 [TN95].

4.2.1 Link-local Addresses

When an interface is configured for IPv6, it must have a *link-local* address. A link-local address is formed by placing a link-local prefix `fe80::` in front of a token, usually the interface's MAC address. In our implementation, this is done by the ifconfig(8) application placing this address on an interface before any other addresses are placed on the same interface. Implementations must be able to detect whether their link-local address has been duplicated on the same link (e.g. Ethernet).[NNS95] Our planned approach to this collision detection is discussed in the Neighbor Discovery section. Once the link-local address is verified as being unique on a link, the first phase of stateless address autoconfiguration is completed. The IPv6 node can then send out ICMPv6 *Router Solicit* messages to locate a router, and begin the second phase of address autoconfiguration.

4.2.2 Router Discovery

IPv6 routers send out periodic *Router Advertisement* messages to the all-nodes multicast address. Also, IPv6 routers send out *Router Advertisement* messages in response to *Router Solicit* messages. Besides performing the traditional jobs of IPv4 router advertisements, IPv6 router advertisements also advertise parameters relating to Neighbor Discovery: suggested MTUs on variable-MTU links, suggested maximum hop limits, and on-link prefixes.

It is the advertisement of on-link prefixes which completes stateless address auto-configuration. If the *Router Advertisement* message indicates that stateless configuration is to be performed, the message will also contain the globally routable address prefix used on the link. The node then takes the token from its link-local address, and prepends the advertised prefix to form an automatically configured globally routable address. The internal code to handle such advertisements also handles the manual address configuration requests from programs such as ifconfig(8).

Unlike IPv4, IPv6 addresses can have lifetimes. In concert with stateless address auto-configuration, lifetimes provide a way for relatively rapid IPv6 address renumbering to occur. Provider-oriented addressing is one of the address schemes that will be used with IPv6.[RLH+95] With provider-oriented addressing, the ability to rapidly renumber many systems at a site is essential if that site should ever want to change network service providers. Hence, IPv6 interface addresses in the kernel now contain lifetime fields.

4.3 Neighbor Discovery

IPv6 does not use ARP.[4] Instead, IPv6 uses multicasting and ICMPv6 to discover the addresses of on-link neighbors.[NNS95] Our implementation uses host routes for on-link neighbors and keeps link-layer information inside the route, much as 4.4BSD implements ARP entries. Like ARP, IPv6 neighbor discovery has the route's gateway address point to a data-link socket address, for example an Ethernet MAC address.

IPv6 Neighbor Discovery is responsible for finding the link address information for the host route entries. If an IPv6 destination is determined to be on link, either by matching an on-link prefix (represented as a cloning network route, as IPv4 does), or by determining that there is no other way to reach a destination, a neighbor solicit is sent out to a special multicast address. The special multicast routing prefix `ff02::1:` is prepended to the low 32 bits of the solicited neighbor. All nodes automatically join the *Solicited Nodes* multicast group appropriate for their own addresses. Broadcast does not exist in IPv6; multicast replaces all uses for broadcast.[5] Once a *Neighbor Solicit* is heard, enough information is known to send a unicast *Neighbor Advertisement* to the solicitor, and now the soliciting node knows that the neighbor is reachable. While the solicted node has enough information to return the unicast neighbor advertisement, reachability the opposite way is not yet confirmed. Unicast solicit and advertisement messages confirm the reachability of the neighbor after initial reachability is established. Upper-level protocols (e.g. TCP) can also be used

[4] Hence, ARP-related broadcast storm problems will not be present with IPv6

[5] Hence, broadcast storms will not exist with IPv6.

to provide reachability confirmation.[6]

Users can use `netstat -r` to examine the state of currently reachable and recently reachable neighbor systems. This neighbor reachability information is kept as part of the routing table in the kernel, so reachability updates for one session to a neighbor will also refresh reachability for other sessions to the same neighbor. Neighbors that have become unreachable will linger in the routing table and will eventually be marked with the `RTF_REJECT` flag. This is similar to the way ARP is handled in 4.4-Lite BSD.

Neighbor discovery can be used to detect the uniqueness of a link-local address. After a link-local address is configured, the node sends a multicast neighbor solicit for it's proposed link-local address. If no neighbor responds with a neighbor advertisement, then the link-local address is unique for the link. The alpha release does not currently implement collision detection, because of the difficulty in placing the functionality of the detection. If done in the kernel, a user process may be trapped in the ioctl(2) call for a long time while collision detection takes place. If done in user space, multiple calls will have to be made into the kernel.

5 Transport Layer Changes

Both the UDP and TCP protocols remain unchanged for IPv6. However, the BSD implementations required modification to provide concurrent support for IPv4 and IPv6. The main difficulties arose due to the different sizes of the IPv4 header and the IPv6 header. Because the TCP and UDP implementations are shared between IPv4 and IPv6, we designed a modified Protocol Control Block (PCB) structure that supports both versions of IP. Had the original BSD implementation of TCP, UDP, and IP not been so closely coupled, it would have been easier to add IPv6 support into the kernel.

5.1 Protocol Control Block

Since TCP and UDP do not change between IPv4 and IPv6, TCP and UDP use the modified Protocol Control Block structures (PCBs) in the same way. With IPv6's larger address space, the PCBs were modified to support both IPv4 and IPv6 addresses and to denote which addresses are actually in use. To support both protocols, new unions were devised. To make these changes invisible to existing code, appropriate `#define`s were added that silently dereferenced the appropriate component of the union. Figure 4 shows an example of a new union and its corresponding new `#define`s.

```
union {
struct route ru_route;
struct route6 ru_route6;
} inp_ru;

#define inp_route inp_ru.route
#define inp_route6 inp_ru.route6
```

Figure 4: Route union used in new PCB structure

The IPv4-IPv6 transition specification [GN95] makes it easier to support both protocols in a single PCB by allocating a portion of the IPv6 address space for use as "IPv4-mapped" addresses, which cannot be used as addresses in IPv6 datagrams. Additionally, if a session is intending to send IPv6 datagrams, a bit in the session's PCB's flags will be set indicating this. If that bit is not set, then IPv4 is in use. The route, IP header template, and multicast options elements now use unions so that either IPv4 or IPv6 can be used with the PCB.

New PCB functions were written to support bind, connect, and notify functions on `PF_INET6` sockets. Because such a socket can be used to send and receive either IPv4 or IPv6 traffic, these functions needed to be separate from the equivalent IPv4 functions and also needed to handle both versions of IP. In the near future we intend to enhance these functions to fully support the IPv6 *Flow Identifier* field so that real-time and predictive services are provided to applications. The `in6_pcbnotify()` function also calls the input security policy function to determine whether a particular error can be passed upwards to the application or whether that would cause a security violation and the error should not be delivered.

5.2 Changes in UDP

The UDP protocol remains unchanged for IPv6, but the BSD implementation needed to be modified to support both versions of IP. The majority of the changes to the UDP code resulted from the need to support the different address format. The changes are minimal and are isolated to the following functions `udp_input()`, `udp_output()`, `udp_ctlinput()`, and `udp_usrreq()`. Almost all changes occur in the input and output processing of UDP datagrams, handled by the functions `udp_input()` and `udp_output()`, respectively.

[6] We are still experimenting with the best way for TCP to update reachability without impairing performance.

Incoming UDP datagrams, regardless of whether they are transported over IPv4 or IPv6, are processed by `udp_input()`. Where the code needs to access elements of the IP header, different code paths are executed for IPv4 and IPv6 datagrams. The function relies on a local variable, which it sets on entrance to the function, to determine which code path to follow. An example of a code path specific to IPv6 is the processing of an IPv4 packet destined for an IPv6 socket. The IPv6 BSD Sockets API specification allows an application to receive both IPv4 and IPv6 datagrams using an IPv6 socket.[GTB95] Code has been added to allow `udp_input()` to handle this special case.

The `udp_input()` function now calls the input security policy function before processing an incoming packet. This ensures compliance with both socket and system security requirements. If an incoming packet should not be delivered for security policy reasons, then it is silently dropped. This check does exact a performance penalty on each received packet, but we have not yet found a better way to handle input security policy checks.

The function `udp_output()` is called to create and send a UDP datagram. It determines whether to create an IPv4 or IPv6 datagram by looking at the protocol control block for the socket originating the datagram. If the socket's protocol family is `PF_INET6` and the socket's PCB indicates that the destination is a native IPv6 address, an IPv6 UDP datagram is composed and sent down to the IP layer via the `ipv6_output()` function. If the protocol family is `PF_INET`, `ip_output()` is called instead of `ipv6_output`. A significant change in `udp_output()` from its IPv4 version involves the calculation of the UDP checksum. In IPv4, calculation of the UDP checksum is optional and is controlled by the global variable `udpcksum`. Since IPv6 no longer has an IP layer checksum, the UDP checksum is not optional and must be calculated for all IPv6 UDP packets. This is necessary to provide integrity protection of the source and destination address that is not provided by IPv6, which lacks an IP header checksum.

The remaining changes in `udp_ctlinput()` and `udp_usrreq()` are minor changes to call IPv6 versions of certain IPv4 functions or to initialize IPv6 specific variables in the protocol control block. Overall, the modifications of UDP code to work with both IPv4 and IPv6 are straightforward.

5.3 Changes in TCP

The TCP protocol also remains unchanged for IPv6, but was modified to support both versions of IP.

One change was to add a new member, `pf`, to the TCP control block structure, `struct tcpcb`. This new member stores the *Protocol Family*, either `PF_INET` for IPv4 or `PF_INET6` for IPv6, in use for each TCP session. This is used in several parts of the TCP code to help select the correct IP-specific code branch.

The beginning of the `tcp_input()` function has a small amount of IP-related processing. This was broken into two code paths, one for IPv4 and one for IPv6 at the cost of an if check and a slight increase in code size.

The main difficulty with the 4.4 BSD-Lite TCP implementation was its reliance on a single pointer, `struct tcpiphdr *ti`, that pointed to a structure containing both the IPv4 overlay header (Figure 5) and also the TCP header of received segments. The `tcp_input()` and `tcp_reass()` functions used this combined structure for most of the data references relating to a given TCP segment. There were also other uses of this structure within the TCP implementation. Because of the differing IP header sizes, the TCP header starts at a different offset from the start of the structure, depending on which IP header is present. The solution to this problem was to create a new pointer `struct tcphdr *th` which is calculated separately for IPv4 and IPv6, but always points to the TCP header. The references to TCP header data that had previously used `*ti` now use `*th` instead.

However, use of the `*th` pointer did not solve all of the problems. The older `struct tcpiphdr` contains an element `ti->ti_len` that pointed to the packet's length field. There is not room to store such a data item in the `struct tcpipv6hdr`, which uses a `struct ipv6ovly` (Figure 6), but fortunately there was an existing local variable `tlen` in `tcp_input()` that is used instead. Most of the references to IP data elements are made at the very beginning of the `tcp_input()` function and so were easily handled.

The `tcp_reass()` function was not amenable to supporting both versions of IP at the same time, so our implementation increases code size by adding a new `tcpv6_reass()` function that uses `struct tcpipv6hdr` in lieu of the `struct tcpiphdr` used by the original `tcp_reass()`.

The `tcp_input()` function now calls the input security policy function before processing an incoming TCP segment. This ensures compliance with both socket and system security requirements. If an incoming segment should not be processed for security policy reasons, then it is silently dropped. If the system security policy is to require authen-

ih_next (pointer to next segment hdr)		
ih_prev (pointer to prev segment hdr)		
ih_x1 (pad)	ih_pr (protocol)	ih_len (length)
ih_src (source address)		
ih_dst (destination address)		

Figure 5: Format of `struct ipovly` IPv4 Overlay

ih_next (pointer to next segment header)
ih_prev (pointer to prev segment header)
ih_src (source address)
ih_dst (destination address)

Figure 6: Format of `struct ipv6ovly` IPv6 Overlay

tication on all received packets, then attempts to open an unauthenticated TCP connection or unauthenticated ping will silently fail as if the destination system were not reachable at all. As with the UDP implementation, this check exacts a performance penalty.

One benefit of our changes has been to isolate the network-layer code more. This might make it easier to modify TCP further to support TCP over other network-layer protocols, for example Novell's IPX. We are concerned about the adverse performance impact of the IPv6 changes, so we are examining methods of improving the performance of our implementation. We have not found anything in the IPv6 specifications that inherently reduces TCP performance.

6 Changes to Applications

6.1 Network Socket Enhancements

Although the IETF does not standardise application programming interfaces, some members of the IPng Working Group did create an Informational RFC describing how IPv6 might be used in conjunction with BSD Sockets [GTB95]. Some changes in 4.4-Lite BSD were needed to comply with that specification. Fortunately, most of the changes involved adding protocol switch tables, and entries to those tables[LMKQ89]. Other sockets changes were implemented at lower levels, most notably the aforementioned PCB code. One can use a `PF_INET6` socket to communicate using IPv4 or IPv6, which makes it easier to transition applications to the new version of IP.

```
#include <sys/socket.h>
#include <netinet6/in6.h>
...
struct sockaddr_in6 addr6;
int s;
...
s = socket(PF_INET6, SOCK_DGRAM, 0);
addr6.sin6_len = sizeof(addr6);
addr6.sin6_family = AF_INET6;
addr6.sin6_port = htons(7);
addr6.sin6_flowinfo = 0;
(void) ascii2addr( AF_INET6,
                   "FE80::800:dead:beef",
                   &addr6.sin6_addr);
sendto( s, ''hello'', 6, 0, &addr6,
        sizeof(addr6));
...
```

Figure 7: Code fragment illustrating use of UDP over IPv6

More extensive changes were needed to permit applications to request security services from IPv6. Several new socket options were defined and implemented, including `SO_SECURITY_ENCRYPTION_TRANSPORT`, `SO_SECURITY_ENCRYPTION_TUNNEL`, and `SO_SECURITY_AUTHENTICATION`. These new socket options are used by an application to request that ESP in transport-mode, ESP in tunnel-mode, or the

Authentication Header be used with this network session. Each also has an associated *Security Level* parameter. There are currently 4 security levels implemented. Level 0 does not use security on outbound packets and does not require it on inbound packets. Level 1 uses security on outbound packets if it is available but does not require it on inbound packets. Level 2 requires security both outbound and inbound. Level 3 is the same as level 2 except that outbound packets use a security association unique to this socket. A planned enhancement is to also permit an application to request that its session be provided with a new security association to replace the one in use. We consider our new security-related socket options experimental and may alter them somewhat as we gain more experience with application issues.

Our kernel implementation permits a system administrator to define a default or minimum level of security. The default security will be used for all sessions provided with a valid Security Association. Applications may also request security services via the above sockets extensions. The system security is configured using the same matrix of 3 protocols and 4 security levels that we described earlier for use in socket-requested security. We plan to enhance the flexibility of our security policy engine in the future so that the system administrator can have more sophisticated policies than are currently supported.

6.2 Key Management Socket

We also have defined a new protocol family, called `PF_KEY`, for the Sockets application programming interface. This extension to Sockets provides a generic interface between security association management applications, such as a Photuris daemon [KS95], and the kernel's network security data structures.[PAM95] This new generic key management interface is modeled upon the existing routing socket, `PF_ROUTE`.[Skl91] This enhancement permits the key management system to be completely decoupled from the IP security implementation. Multiple key management schemes can be supported concurrently if desired. It also will make it easy to change from one key management algorithm or protocol to a new key management algorithm or protocol. To make such a change, only a new daemon needs to be installed; no kernel modifications or kernel rebuilding is necessary. Many published key management protocols have had flaws discovered years after initial publication[NS78][DS81]. Hence it is important to be able to easily change the key management protocol being used by the system. Our alpha release includes an application, *key(8)*, that can be used by the system administrator to manage keys and security associations in the kernel. Any key management scheme, whether automatic key management such as Photuris or manual key management such as *key(8)*, can use the `PF_KEY` interface.

6.3 An Example Application: telnet

Most applications will need a small amount of modification to take advantage of IPv6 and its unique features. Even with these modifications, the applications will continue to support IPv4. Most of these modifications are in the socket code, allowing the use of the new `AF_INET6` address family, new data structures, and the corresponding network functions.

We have modified several applications to use IPv6. We describe the modifications required for telnet in the following paragraphs. The telnet application was also enhanced to add command-line options to set the socket security level. [7]

The telnet client first parses the command line and options. If the user has requested IP security services, then the appropriate socket options are set using `setsockopt()`. Telnet then uses the new `hostname2addr()` and `ascii2addr()` functions to seek an IPv6 address for the specified hostname or text representation of an address. If an IPv6 address is returned, telnet then opens a `PF_INET6` socket and begins communicating. The requested security services are automatically applied by the IP security implementation inside the kernel. If an IP security processing error (for example, no security association can be found and one is needed) occurs, then the `EIPSEC` error will be returned to telnet so the user can be informed of the problem.

The IPv4 library functions `inet_ntoa()`, `inet_aton()`, `gethostbyname()`, and `gethostbyaddr()` have been superceded by the new library functions `addr2ascii()`, `ascii2addr()`, `hostname2addr()`, and `addr2hostname()` [GTB95].[8] These new library functions work equally well for both IPv4 and IPv6, making it easier for applications to support both IPv4 and also IPv6.

In the future, we plan to add a privileged socket option to permit applications that need to bypass IP security to do so (for example, a Photuris daemon). This socket option would fail if the effective user-id of the process connected to the socket was not equal

[7] Although 4.4 BSD's telnet includes an encryption option, a fatal implementation flaw limits its practical value.

[8] These new functions were originally suggested by Craig Partridge in an email note to the IETF's IPng mailing list.

to 0 so that ordinary user applications could not bypass system security. Such bypass is needed by key management applications so that they can create the initial security associations. Certain other applications having application-layer security, for example a secured Domain Name Service daemon, might also need to bypass IP security services. Although this has not been implemented yet, we believe it will be straight forward to implement and have already put some of the hooks in place.

7 Performance

Throughput and round-trip latency were measured using Rick Jones' NetPerf tool.[Jon95] NetPerf has more accuracy and reproducibility than some older tools.[Jef95] Except for Table 5, these measurements are for traffic that is neither authenticated nor encrypted, though the security policy checks are still performed.

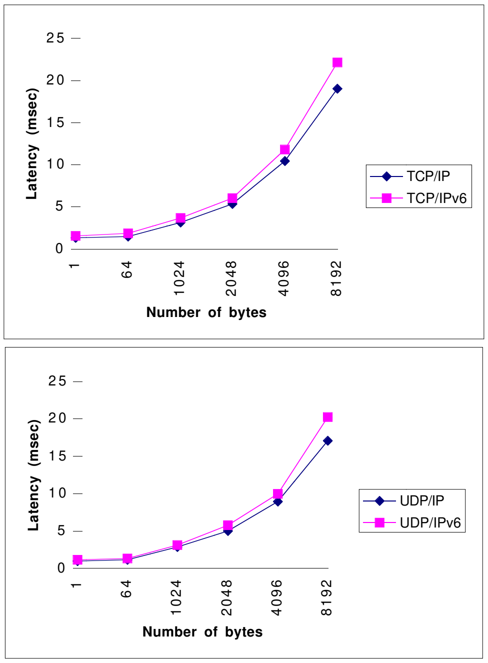

Figure 8: UDP and TCP Latency Graphs

In our alpha release, IPv6 performance is somewhat worse than IPv4. UDP latency, shown in Table 2, and TCP latency, shown in Table 1, both increased for IPv6. The increased latency, shown in Figure 8 is in both the inbound and outbound protocol processing. Comparing longer addresses (four 32-bit words vs. a single 32-bit word) and preparsing of optional headers are the major contributors to the increased latency. We plan to add a fast path bypass

Number of bytes.	IPv4. (msec).	IPv6. (msec).	Percent increase.
1	1.27	1.54	+21%
64	1.45	1.83	+26%
1024	3.12	3.62	+16%
2048	5.34	6.01	+12%
4096	10.4	11.9	+14%
8192	19.0	22.1	+16%

Table 1: TCP Latency

Number of bytes.	IPv4. (msec).	IPv6. (msec).	Percent increase.
1	0.93	1.08	+17%
64	1.13	1.30	+15%
1024	2.82	3.06	+8%
2048	5.00	5.77	+15%
4096	8.89	9.90	+11%
8192	17.0	20.2	+19%

Table 2: UDP Latency

around the preparsing code in the future. The lower IPv6 throughput, shown in Table 3 and Table 4, is due to increased latency and larger packet size.

The 4.4-Lite BSD implementation of TCP/IPv4 has had years of optimisation whilst our alpha release has had no optimisation. We believe that an optimised IPv6 implementation will perform at least as well as a similarly optimised IPv4 implementation.

Data size	Socket buffer size	IPv4 (KB/sec)	IPv6 (KB/sec)	Perf. drop
4096	57344	780	731	6.26%
8192	57344	778	729	6.28%
32768	57344	776	730	5.97%
4096	32768	807	763	5.45%
8192	32768	806	758	5.91%
32768	32768	811	762	6.02%
4096	8192	861	775	9.93%
8192	8192	858	784	8.68%
32768	8192	863	784	9.19%

Table 3: TCP Throughput

NetPerf has not yet been modified to use the security socket options. Making such modifications to NetPerf does not appear trivial. The older *ttcp(8)* testing tool was easily modified to use the security socket options. Table 5 indicates throughput differences (measured with *ttcp(8)*) using authentication, transport-mode encryption, and both, versus no security at all. While we have less confidence in the

Data size	Socket buffer size	IPv4 (KB/sec)	IPv6 (KB/sec)	Perf. drop
64	32767	537	500	6.82%
1024	32767	1144	1125	1.60%

Table 4: UDP Throughput

absolute values for *ttcp(8)* than for NetPerf, we believe the relative performance degradation shown by *ttcp(8)* is meaningful. Our security implementations have not been optimised at all. We believe that we can noticably improve our encryption performance by encrypting and decrypting in place and removing memory copies. Hardware implementations of DES that run at 1 Gbps exist.[Sch94] Implementations seeking high performance should probably use such encryption hardware.

Security Features	Throughput (KB/sec)
None	~775
Authentication	~345
Encryption	~192
Both	~153

Table 5: Impact of IPv6 Security On Throughput.

8 Summary

This paper has described a freely distributable prototype implementation of IPv6 based on 4.4 BSD-Lite. There are a number of implementation differences between IPv4 and IPv6 due to packet format differences and also protocol differences. Some of the assumptions made and techniques used by the IPv4 implementation are no longer valid for IPv6. Because the implementation includes the cryptographic security mechanisms mandatory for IPv6, any networked application can now have the security it desires without having to implement it at the application layer. Performance of TCP/IPv4 and TCP/IPv6 has been compared.

9 Acknowledgments

This work has been funded by the Information Security Program Office (PD71E) of the US Space & Naval Warfare Systems Command since 1992 and also by the Computer Systems Technology Office of the Advanced Research Projects Agency (ARPA/CSTO) since 1995. We are grateful for their support.

References

[Atk95a] Randall Atkinson. IP Authentication Header, August 1995. RFC-1826.

[Atk95b] Randall Atkinson. IP Encapsulating Security Payload (ESP), August 1995. RFC-1827.

[Atk95c] Randall Atkinson. IP Security Architecture, August 1995. RFC-1825.

[DC95] Steve Deering and Alex Conta. ICMP for the Internet Protocol version 6, June 1995. Work in Progress.

[Dee89] Steve Deering. Host extensions for IP Multicasting, August 1989. RFC-1112.

[Dee91] Steve Deering. ICMP Router Discovery Messsages, September 1991. RFC-1256.

[Dee93] Stephen E. Deering. SIP: Simple Internet Protocol. *IEEE Networks*, 7(3):16–28, May 1993.

[DH95] Steve Deering and Bob Hinden. IPv6 specification, June 1995. Work in Progress.

[DS81] D.E. Denning and G.M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.

[FMMT84] R. Finlayson, T. Mann, J. Mogul, and M. Theimer. Reverse address resolution protocol, June 1984. RFC-903.

[GN95] Robert E. Gilligan and Erik Nordmark. Transition Mechanisms for IPv6 Hosts and Routers, May 1995. Work in Progress.

[GTB95] Robert Gilligan, Susan Thomson, and Jim Bound. IPv6 Program Interfaces for BSD Systems, July 1995. Work in progress.

[Hin94] Robert Hinden. Simple Internet Protocol Plus white paper, October 1994. RFC-1710.

[Jef95] Jeffrey D. Chung and C. Brendan and S. Traw and Jonathan M. Smith. Event-Signaling within Higher Performance Network Subsystems. In *Proceedings, High Performance Communications Subsystems*, Mystic, CT, August 1995.

[Jon95] Rick A. Jones. NetPerf: A Network Performance Benchmark (Revision 2.0), February 1995. Technical Report.

[KS95] Phil Karn and William Simpson. The Photuris Session Key Management Protocol, October 1995. work in progress.

[LM91] X. Lai and J. Massey. A Proposal for a New Block Encryption Standard. In *Advances in Cryptology – EUROCRYPT '90 Proceedings*, pages 389–404, Berlin, 1991. Springer-Verlag.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation or the 4.3 BSD UNIX Operating System.* Addison-Wesley, New York, NY, 1989.

[Lot92] Mark Lottor. Internet Growth (1981-1991), January 1992. RFC-1296.

[MD90] Jeff Mogul and Steve Deering. Path MTU Discovery, November 1990. RFC-1191.

[MKS95a] Perry Metzger, Phil Karn, and William Simpson. The ESP DES-CBC transform, August 1995. RFC-1829.

[MKS95b] Perry Metzger, Phil Karn, and William Simpson. IP Authentication using Keyed MD5, August 1995. RFC-1828.

[NNS95] Erik Nordmark, Thomas Narten, and William Simpson. Neighbor Discovery for IP Version 6, September 1995. Work in Progress.

[NS78] R.M. Needham and M.D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[PAM95] Bao G. Phan, Randall J. Atkinson, and Daniel L. McDonald. PF_KEY: Key Management Support inside 4.4 BSD Unix, December 1995. Technical Report.

[Plu82] D. Plummer. Ethernet address resolution protocol, November 1982. RFC-826.

[Pos81] Jon Postel. Internet Control Message Protocol, September 1981. RFC-792.

[RLH+95] Yakov Rekhter, Peter Lothberg, Robert Hinden, Steve Deering, and Jon Postel. An IPv6 Provider-Based Unicast Address Format, August 1995. Work in Progress.

[Sch94] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, New York, NY, 1994.

[Skl91] Keith Sklower. A Tree-Based Packet Routing Table for Berkeley UNIX. In *Proceedings of the Winter '91 USENIX Conference*, Dallas, TX, January 1991. USENIX Association.

[TN95] Susan Thomson and Thomas Narten. IPv6 Stateless Address Autoconfiguration, October 1995. Work in Progress.

[ZBE+93] L. Zhang, R. Braden, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Networks*, September 1993.