USENIX Association

# Proceedings of the BSDCon 2002 Conference

San Francisco, California, USA
February 11-14, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

# Flexible Packet Filtering: Providing a Rich Toolbox

Kurt J. Lidl
*Zero Millimeter LLC*
*Potomac, MD*
kurt.lidl@zeromm.com

Deborah G. Lidl
*Wind River Systems*
*Potomac, MD*
deborah.lidl@windriver.com

Paul R. Borman
*Wind River Systems*
*Mendota Heights, MN*
paul.borman@windriver.com

## Abstract

The BSD/OS IPFW packet filtering system is a well engineered, flexible kernel framework for filtering (accepting, rejecting, logging, or modifying) IP packets. IPFW uses the well understood, widely available Berkeley Packet Filter (BPF) system as the basis of its packet matching abilities, and extends BPF in several straightforward areas. Since the first implementation of IPFW, the system has been enhanced several times to support additional functions, such as rate filtering, network address translation (NAT), and traffic flow monitoring. This paper examines the motivation behind IPFW and the design of the system. Comparisons with some contemporary packet filtering systems are provided. Potential future enhancements for the IPFW system are discussed.

## 1 Packet Filtering: An Overview

Packet filtering and packet capture have a long history on computers running UNIX and UNIX-like operating systems. Some of the earliest work on packet capture on UNIX was the CMU/Stanford Packet Filter [CSPF]. Other early work in this area is the Sun NIT [NIT] device interface. A more modern, completely programmable interface for packet capture, the Berkeley Packet Filter (BPF), was described by Steve McCanne and Van Jacobson [BPF]. BPF allows network traffic to be captured at a network interface, and the packets classified and matched via a machine independent assembly program that is interpreted inside the kernel.

### 1.1 BPF: An Overview

BPF is extremely flexible, machine independent, reasonably high speed, well understood, and widely available on UNIX operating systems. BPF is an interpreted, portable machine language designed around a RISC-like LOAD/STORE instruction set architecture that can be efficiently implemented on modern computers.

BPF only taps network traffic in the network interface driver. One important feature of BPF is that only packets that are matched by the BPF program are copied into a new buffer for copying into user space. No copy of the packet data needs to be made just to run the BPF program. BPF also allows the program to only copy enough of a packet to satisfy its needs without wasting time copying unneeded data. For example, 134 bytes is sufficient to capture the complete Ethernet, IP, and TCP headers, so a program interested only in TCP statistics might choose to copy only this data.

A packet must be parsed to determine if it matches a given set of criteria. There are multiple ways of doing this parsing, but a great deal of it amounts to looking at a combination of bits at each network layer, before the examination of the next layer of the packet. There are multiple data structures designed for efficient representation of the parsing rules needed to classify packets. BPF uses a control flow graph (CFG) to represent the criteria used to parse a packet. The CFG is translated into a BPF machine language program that efficiently prunes paths of the CFG that do not need to be examined during the parsing of a packet.

Ultimately, a standard BPF program decides whether a packet is matched by the program. If a packet is matched by the program, the program copies the specified amount of data into a buffer, for return to the user program. Whether or not the packet was matched, the packet continues on its normal path once the BPF program finishes parsing the packet.

BPF also has a limited facility for sending packets out network interfaces. BPF programs using this facility must bind directly to a particular network interface, which requires that the program know what interfaces exist on the computer. This allows for sending any type of network packets directly out an interface, without regard to the kernel's routing table. This is how the `rarpd` and `dhcpd` daemons work on many types of UNIX computers.

BPF, as originally described, does not have a facility for rejecting packets that have been received. BPF, although described as a *filter*, can match packets, copy them into other memory, and send packets, but it cannot drop or reject them.

## 2 Motivation

The need for a powerful and flexible packet matching and filtering language had been evident for a long time. The basic ideas for the BSD/OS IPFW system were the result of several years of thought about what features and functions a packet filtering system must provide. Having highly flexible packet filtering for an end system would be mandatory, and that same filtering system should be applicable for filtering traffic that was being forwarded through a computer.

The immediate need for a flexible packet filtering framework came from a desire to run an IRC client in a rigidly controlled environment. This environment consisted of a daemon that could be run in a chroot'd directory structure, as well as a highly restrictive set of packet filters. These filters could not just prevent unwanted inbound packets, but perhaps more importantly, could also discard unwanted outbound packets. The BSD/OS IPFW system was thus originally intended to filter both the inbound and outbound traffic for a particular host.

Many of the most popular contemporary packet filtering systems of the initial design era (circa 1995) were incapable of filtering packets destined for the local computer or originating from the local computer. The available filtering systems concentrated on filtering traffic that was being forwarded through the computer. Other major problems with the existing packet filtering systems were the inability to do significant stateful packet forwarding and unacceptably low performance [screend]. `screend` does keep track of IP fragments, which is a limited form of stateful packet filtering.

Further motivation for a flexible packet filtering system was the lack of any other standard packet filtering in the stock BSD/OS system of that era. There was customer demand for a bundled packet filtering system, which was not fully met by the other widely available packet filtering systems [ipfilter]. `screend`, the most widely available contemporary packet filtering system, provided many good lessons in packet filtering technology. The BSD/OS IPFW system was designed with the lessons learned from `screend` in mind.

A consideration for the implementation of a new, flexible packet filtering framework was the realization that as the Internet grew, the number of attacks from other locations on the Internet would also grow. Having a powerful matching language tied to the filtering capabilities would allow for a single BSD/OS computer acting as a router to protect any other computers behind the filter.

## 3 Need for Flexibility

An early design decision for IPFW was that the system should present as flexible a matching and filtering framework as possible. As few filtering rules as possible should be directly embedded in the kernel. As much as possible, filtering configuration and policy should be installed into the kernel at runtime, rather than compiled into the system. This decision has reaped many benefits during the lifetime of this system. Because IPFW is extremely flexible, it has been applied to many problems that were not in mind at the time it was designed. To borrow Robert Scheifler's quote about the X Window System Protocol, IPFW is "intended to provide mechanism, not policy." [RFC1013]

## 4 Other Packet Filters

As was noted earlier, there were several other packet filtering technologies when IPFW was first envisioned. In the years since, other filtering technologies have been developed, some specific to a particular operating system and others available on a variety of platforms. A comparative analysis with these other packet filters allows one to more fully appreciate the flexibility of IPFW.

One of the most important differences between IPFW and these other filtering systems is that IPFW actually downloads complete programs to be evaluated against the packets. The other filtering systems are all rules-based. By evaluating an arbitrary program, an entirely new methodology of packet filtering can be installed without rebooting the system. In a rules based system, any new type of rules requires code changes to the filtering system, as well as a reboot to make it active. Dynamic loadable kernel modules can approximate the program download facility as modules could be replaced with new filtering rule capabilities without requiring a system reboot.

### 4.1 Darren Reed's ipfilter

The ipfilter package is available on many versions of many UNIX-like operating systems, from BSD/OS to older systems such as IRIX to small-footprint systems like QNX to frequently updated systems like FreeBSD. It supports packet filtering, provides a Network Address Translation (NAT) implementation, and can perform stateful packet filtering via an internal state table. Like the other examined packet filters, ipfilter is a rules based system. It can log packet contents to the pseudo-device "ipl." [ipfilter] [ipfilterhowto]

### 4.2 FreeBSD's ipfirewall System

FreeBSD provides a packet filtering interface, known as ipfirewall. This system is often referred to as `ipfw`, which is the name of the management command.[1] This is a rules based packet filtering mechanism, which is manipulated internally by socket options. There is an additional kernel option (`IPDIVERT`) to add kernel divert sockets, which can intercept all traffic destined for a particular port, regardless of the destination IP address inside the packet. The divert socket can intercept either incoming or outgoing packets. Incoming packets can be diverted after reception on an interface or before next-hop routing. [FreeBSD]

### 4.3 Linux 2.2: ipchains

The Linux ipchains implementation provides three different services: packet filtering, NAT (called masquerading), and transparent proxying. The packet filtering capabilities are based on having "chains" of rules, which are loaded at three different filtering locations: input, forward and output. Each "chain" location can have multiple rules appended, inserted or deleted from that location. The rules are relatively simple and allow for chaining to another named rule if a particular criteria is matched. Arbitrary data inspection of packets is not permitted. [ipchains]

### 4.4 Linux 2.4: iptables

The Linux iptables implementation (sometimes referred to as "netfilter") is a complete rewrite and extension of the ipchains filtering system. Substantial cleanup and fixing of multiple idiosyncrasies in handling how packets destined for the local computer are processed have been made. Support for stateful packet filtering has also been added to the system. The command line syntax for specifying packet headers for each rule has been changed since the ipchains release. A `QUEUE` disposition for a packet has been added, which specifies that the packet will be transferred to a user process for additional processing, using an experimental kernel module, `ip_queue`. [iptables]

### 4.5 OpenBSD's "pf" System

OpenBSD 3.0 includes pf, a packet filter pseudo-device. As a rules-based filter, users are restricted to the available set of rules included with pf. Manipulation of the pf pseudo-device is managed through the `pfctl` command. Internally, the system is controlled by `ioctl` calls to the pf device. Rules can be applied on an *in* or *out* basis, and can be tied to a specific interface as well. As a very new packet filtering mechanism (it was written from scratch, starting in June 2001) it does not have an established track record, and is still undergoing change. [OpenBSD]

### 4.6 TIS Firewall Toolkit

The TIS Firewall Toolkit (fwtk) and other proxy firewalls not only examine the source and destination of packets, but also the protocol being sent. New application proxies that understand the protocol must be written for each new type of service. While this approach does allow for additional levels of security as the proxy can watch for attack methods that exploit a particular protocol, it requires a much deeper understanding of each new protocol before filtering that type of traffic. [fwtk]

## 5 Design Elements

Several elements of the overall design and implementation of the BSD/OS IPFW system are worth a detailed examination. Some of the more interesting design choices are discussed below.

### 5.1 BPF Packet Matching Technology

Because of the many fine matching properties of BPF system, as noted in Section 1, it was selected as the core technology for packet matching and classification in the BSD/OS IPFW system.

### 5.2 Download Filter Programs into Kernel

The concept of downloading filters into the kernel was not a novel idea. The IPFW author was familiar with a few obscure packet filter technologies that had the filter coded directly into the network stack.[2] While highly inflexible in operation, this type of filter system did make an attacker work harder when attempting to subvert or weaken an installed filter. The marginal security benefit of a filter compiled into the kernel was dwarfed by the numerous advantages of a downloadable packet filter. Early versions of IPFW had the ability to both password protect filters as well as make downloaded filters immutable. Both of these features were eventually dropped as the additional security provided only came into effect once the computer running the filter was compromised. Once the computer has been compromised to that extent, the added security was not considered to be valuable enough to warrant the costs of maintaining the implementation.

### 5.3 IPFW Kernel Socket

Prudent reuse of kernel facilities is always a goal when designing a new subsystem for the UNIX kernel. The BSD/OS IPFW system needed a method for transmitting data about packets and filter programs from the kernel to programs running in userspace. In some other historic packet filters, this would have been done via the `ioctl` system call, which requires some artificial file to open. Adding a new system call for this purpose might be justified, but every new system call is generally viewed with suspicion.

Instead of adding a new system call, a new instantiation of a kernel socket was made. A new pseudo-IP protocol was defined, which is accessed via a raw internet domain socket. Because sockets were defined to provide an efficient mechanism of moving streams or packets of data to and from the kernel, they are appropriate for the task of moving data about packet filtering to a user application. In the case of the IPFW socket, the data is always generated by the kernel.

The raw IPFW socket provides important functionality in a standard interface with which programmers are familiar. The socket interface also provides for zero (or many) readers of the data. An IPFW filter can send packets or data about packets it has matched back to a userspace program, regardless of the final disposition of the packet. The userspace program may then log the packet, or it might further process a rejected packet, including re-insertion of a possibly modified packet back into the network via a raw IP socket.

High precision timestamps, in the form of a `timespec` structure, are available on packets read from the kernel socket, if the user has requested them. This timestamp is added during the logging operation, so the user application does not have to worry about getting an accurate timestamp when it reads the packets from the socket.

IPFW uses the `sysctl` system call to pass information about filter programs back and forth between the kernel and userspace programs. `sysctl` is used to copy the filter programs into the kernel as they are installed. It is also used for gathering statistics about the IPFW filters installed on the computer. The `sysctl` interface is another example of a flexible programming paradigm. It provided a natural expression of hierarchy that was easily expandable, did not require artificial files to open and reused an existing kernel interface.

## 5.4 Multiple Filtering Points

One of the unique features of the BSD/OS IPFW system at the time it was designed was the inclusion of multiple filtering points in the kernel. The original BPF system only allowed for tapping of packet traffic at each physical interface. The BSD/OS IPFW system provides five logical points where filters may be installed in the kernel.

Table 1 lists each filtering location in the kernel. Each filtering location has an associated default action. When a filtering location has at least one filter installed, if no explicit disposition for a packet is provided by the filter, the default action will be applied to the packet. The passage of packets through the various filtering locations is described in detail in Section 6 of this paper.

| *Location* | *Modify?* | *Default Action* |
|---|---|---|
| pre-input | yes | accept |
| input | no | reject |
| forward | yes | reject |
| pre-output | yes | accept |
| output | no | reject |

Table 1: IPFW Filtering Locations

## 5.5 Stackable Filters

Each filtering point in the kernel is actually the attachment point for a stack of filter programs. Filter programs can easily be pushed onto the stack, popped off the stack, or inserted into the middle of the stack for each filtering point. Individual filters each have a priority (a signed 32 bit number) that determines where the in the stack the filter is actually placed. Multiple filters installed at the same priority, at the same filtering location, operate as a traditional stack.

Filters may also have a symbolic tag to aid in their identification, replacement, or deletion.

## 5.6 Flexibility of Actions

After classifying a packet according to whatever rules are in place, a packet filtering system has to perform an operation on the packet. A simple packet filtering system has just two operations, “accept” and “reject.” The BSD/OS IPFW system has three additional operations. The `log` action takes a specified amount of the packet and copies it to the IPFW kernel socket. The `call` action allows the current packet to be passed to a different named filter for further processing. The `next` action calls the next filter in the stack of filters installed at the current filter location. In addition, the BSD/OS IPFW system allows packets to be modified explicitly by the filter program, or as the consequence of calling another filter program. The classic “accept” and “reject” actions have been extended so they can also optionally log the packet to the kernel socket.

## 5.7 Filter Pool

In addition to the explicit filtering points in the kernel a pool of filter programs can be installed into the kernel, not associated with a particular filtering point. This allows common filter programs to be installed into the filter pool and then be referenced from any of the other filters installed in the running system. Currently only BPF based filters have the ability to call a filter from the pool. The filter called may delete the packet or return a value associated with the packet. Typically this value is boolean. The called filter might also be used to record some state that can later be accessed.

Unlike BPF programs, it is possible to create an infinite loop of called filters. There is no loop detection in the filter software, which could be considered a flaw. Users of the IPFW system are obligated to understand the interactions between all their filter programs.

### 5.8 Circuit Cache

Although BPF filters themselves are stateless, by using custom coded filters, such as the circuit cache, the filters can access saved state about a connection. The circuit cache provides the system with two features. The first is the ability of a BPF program to request the circuit described by a packet be added to the cache. A circuit is defined as the combination of the source and destination addresses, along with the source and destination ports for the upper level protocol, if relevant. The second is the ability to pass a packet to the cache for it to determine if that session has been seen before. For example, TCP packets can be divided into "Initial SYN" packets and "Established" packets. Initial SYN packets are subject to potentially complicated rules to determine if the session should be allowed. If the packet is to be accepted, it is passed to the circuit cache asking for an entry to be added for its circuit. Any Established packet is simply passed to the circuit cache for query. If the packet does not match an existing session, it is rejected. The circuit cache understands the TCP protocol and when caching TCP circuits it can optionally monitor FIN and RST packets and automatically terminate a circuit when the TCP session is shut down. Circuits may also automatically be timed out to reclaim kernel resources after a configuration period of inactivity.

### 5.9 Custom Coded Filters

While BPF-based filters are the most flexible and commonly used filters within the BSD/OS IPFW system, they are not the only method of defining a filter. There are a variety of custom coded filters available. Custom coded filters are C modules that are compiled into the kernel. These typically provide a very rigid set of filtering capabilities. Some non-BPF filters included with IPFW can be used to write traditional, rules based filters. These non-BPF filters may not be able to make use of the advanced features of IPFW due to limitations in their design. Several examples of custom coded filters are described in Section 10 of this paper.

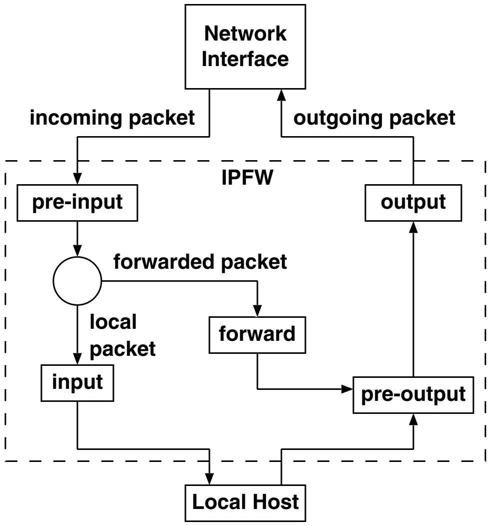

Figure 1: IPFW Filtering Locations

### 5.10 Transparent Proxying

IPFW's ability to force any packet to be delivered to the local computer allows for the creation of transparent proxies for multiple services. An additional small change to the TCP stack in BSD/OS complements this ability. The SO_BINDANY socket option allows a program to listen on a particular port, and bind to whatever IP address for which the connection request was originally intended. This happens regardless of whether the IP address is bound to one of the computer's interfaces. This support makes writing transparent proxies straightforward.

## 6 How it Works

IPFW operates on Internet Protocol (IP) packets that are received or sent by the computer running IPFW. In general there are three types of packets: packets that were sent to the computer, packets that were generated by the computer, and packets for which the computer is acting as a forwarder.

When a packet arrives on the computer (the packet is either sent to this computer or this computer is forwarding the packet), the network driver copies that packet into an mbuf. If the packet is an IP packet, it is placed on a queue of IP packets to be processed by the kernel. The interface on which the packet arrived is also recorded in the mbuf and can be retrieved by any called IPFW filter.

The ip_input() routine in the kernel then dequeues the packet, performs sanity checks on the packet and determines the destination for the packet. If the destination is the local computer, the kernel will perform packet reassembly. IP packets may be broken into smaller packets (fragmented) if a network element in the path between the source and destination is not able to handle the entire packet as a single datagram. Finally, once the packet is complete, the kernel will queue the packet on the correct IP protocol queue (such as TCP or UDP). Packets that are to be forwarded are not re-assembled. These packets are sent on to ip_forward() and eventually on to ip_output() for transmission to the destination.

When IPFW is used, ip_input() will call the pre-input filter chain, if present, just after performing basic sanity checks. This filtering is performed prior to determining the destination of the packet. Because very little examination of the packet has been performed, and no extra state about the packet is stored in the kernel, it is safe for the IPFW filter to modify the packet contents. It may even force a packet to be delivered to the local computer, even if the destination address does not match the address of any of the interfaces on the computer. The most basic modification is to delete the packet, which causes ip_input() to stop processing the packet. Allowing modification of any type at this point allows for various specialty filters such as NAT and packet reassembly. Packet reassembly can be performed explicitly by calling the "rewrite" named filter. Packet reassembly is useful so that following filters will always see complete IP packets and not IP fragments. The ability to modify the packet is the reason that the pre-input filter point was added to IPFW.

Once any filters on the pre-input filter point have been executed, ip_input() continues with normal processing, which is to determine the destination of the packet. If the packet is to be delivered locally, then processing continues normally up until the point where the packet would be queued for an upper level protocol. The fully formed packet is passed to the input filter chain. No modification of the packet contents is allowed at this point as significant sanity checks have been performed on the packet. The packet may still be dropped, logged, or both dropped and logged. Once the packet completes the input filter processing, it is either discarded (rejected) or queued for an upper level protocol as normal.

Received packets that are not to be delivered locally are to be forwarded and are passed to ip_forward(). The ip_forward() routine determines if the packet can be forwarded by the computer. This decision is made by ensuring a route exists for the destination address and the packet's time to live has not expired. The packet is then passed to the forward filter chain. The forward filters have access to the interface indexes for both the input and probable output interfaces. It is possible for the output interface to change between ip_forward() and ip_output(), though typically this is not the case. Knowledge of the input and output interfaces provides assistance in filtering packets with spoofed addresses. The forward filter, like the pre-input filter, is allowed to modify the packet. The main restriction on modifications is that a forwarded packet should not be modified into a local packet. The packet should either still be destined for an external computer after modification or it should be deleted. Once the forward filter chain has been called the rest of ip_forward() is executed and eventually the packet is passed on to ip_output().

Packets passed to ip_output() are either locally generated or being forwarded through this computer. In both cases, ip_output() verifies that a route exists for the destination address and (re)determines the destination interface for the packet. The pre-output filter chain is then called. This filter, much like the pre-input filter, may modify the packet. In addition, it may specify a different IP address to be used for the next-hop routing of this packet. This override of the next-hop routing destination is done through an out-of-band mechanism. This capability allows the pre-output filter to actually determine which interface the packet should be sent out when there are multiple possible output interfaces. If an IP address is provided via the out-of-band method, or the destination IP address inside the packet is changed, the routing lookup is repeated. The pre-output filter is not called a second time.

For forwarded packets, all filtering is now complete. For packets that were locally generated the output filter chain is called immediately after the pre-output filter. Like the input filter chain, the packet may not be modified by the output filter chain. The ip_output() routine will eventually call the network interface's output routine. If IPFW rate filtering (as discussed in Section 10) is being used, the ip_rateoutput() routine is actually called instead of the interface's output routine. The ip_rateoutput() routine is responsible for eventual delivery of the packet to the network interface or dropping of the packet.

## 7 BPF Language Overview

The most used and most flexible filter type in IPFW is the BPF filter. As mentioned earlier, this type of filter uses the BPF pseudo-machine. The BPF pseudo-machine has been enhanced for use with IPFW. Only one totally new BPF instruction was added for IPv4 packet processing. A new memory type was added, as well as the ability to modify the packet being processed. IPv6 enhancements have been added and are discussed at the end of this section.

The new BPF instruction, CCC, enables the calling of a filter on the "call filter chain." While it might seem that the acronym stands for "Call Call Chain," it was actually derived from "Call Circuit Cache." The circuit cache was the reason for the creation of the call chain. The CCC instruction returns the result of the call in the A register.

The new memory type is called ROM and is an additional memory area to the original BPF memory spaces. The original memory spaces included the packet contents as well as the scratch memory arena. While the first implementation did in fact store read only information, the term ROM is now a misnomer as the ROM locations can be modified by the filter. This space, called "prom" in the source code, is used to pass ancillary information in and out of the BPF filter.

While the bpf_filter() function does not have any innate knowledge of the meaning of these memory locations, IPFW assigns meanings to several locations:

| | | |
|---|---|---|
| 0 | IPFWM_AUX | An auxiliary return value (for errors) |
| 1 | IPFWM_SRCIF | The index of the source interface (if known) |
| 2 | IPFWM_DSTIF | The index of the destination interface (if known) |
| 3 | IPFWM_SRCRT | The index of the interface for return packets |
| 4 | IPFWM_MFLAGS | The mbuf flags |
| 5 | IPFWM_EXTRA | Bytes of wrapper that preceeded this packet |
| 6 | IPFWM_POINT | What filter point was used |
| 7 | IPFWM_DSTADDR | New address to use for routing to destination |

The BPF filter is intelligent about setting these values. As some of these values, such as IPFWM_SRCRT, can be expensive to calculate, the filter is examined when passed into the kernel. A bitmap is built of all ROM locations referenced by the program and only those locations are initialized.

In order to support the ROM memory space, the calling convention of the bpf_filter() function was changed to pass three additional parameters:

```
int32_t *prom; /* ptr to ROM memory */
int promlen;   /* count of valid bytes */
               /* in the memory space */
int modify;    /* boolean to indicate */
               /* whether packet */
               /* can be modified */
               /* by bpf_filter() */
```

All existing calls to bpf_filter() were modified to pass NULL, 0, 0 for these three values.

IPFW has been adapted for use with IPv6. This work was implemented with the NRL version of IPv6. More recent releases of BSD/OS use the KAME IPv6 implementation. The changes to support IPFW in the KAME IPv6 stack have not yet been written.

In order to support IPv6, several other new enhancements were made to the BPF pseudo-machine. Triple length instructions were added. A "classic" BPF instruction is normally 64 bits in size: 16 bits of opcode, two 8 bit jump fields, and a 32 bit immediate field. A triple length instruction has 128 bits of additional immediate data (the length of an IPv6 address). A new register, A128, was also added. The load, store, and jump instructions now have 128 bit versions. The scratch memory locations have been expanded to 128 bits, though traditional programs only use the lower 32 bits of each location. An instruction to zero out a scratch memory location (ZMEM) was added. Because BPF was not extended to handle 128 bit arithmetic, a new jump instruction was created that allowed for the comparison of the A register to a network address, subject to a netmask. The netmask must be specified as a CIDR style netmask, specifically a count of the number of significant bits in the netmask.

ROM locations only have 32 bit values and it is in the ROM that a new destination routing address is passed. Currently it is not possible to use the next-hop routing capability with IPv6.

## 8 IPFW Filtering Language

Initially BPF filters were written in BPF assembly[3] with the aid of the C pre-processor (cpp). It was thought that many assembly fragments would be written for various needs and that the final filter would include these fragments. It was quickly determined this was not a very user friendly way of programming filters. It yielded opaque filters such as:

```
// IP header length into X
ldx  4 * ([0] & 0xf)
// Protocol of packet
```

```
    ld   [9 : 1]
    // Is it UDP?  Jump to L1 if not
    jeq  #17 - L1
    // Move ip length into A
    txa
    // Add 8 bytes to skip UDP header
    add  #8
    jmp  L3
L1:
    // Is it TCP?  Jump to L2 if not
    jeq  #6 - L2
    // Load TCP flags into A
    ld   [x + 13 : 1]
    // Jump to L11 if SYN bit is set
    jset #2 L11 -
    // If SYN is not set, just accept it
    ret  #IPBPF_ACCEPT
L11:
    // Move ip length into A
    txa
    // Add 20 bytes to skip TCP header
    add  #20
    jmp  L3
L2:
    // Just move ip header length into A
    txa
L3:
    or   #(IPBPF_ACCEPT | IPBPF_REPORT)
    // Accept the packet and report it
    ret  A
```

A new language was clearly needed.[4] Existing filtering languages were of little help as they were rules based and not programmatic. The ability to use the programable features of BPF was a key design goal of IPFW. Since BPF does not allow reverse jumps, there is no facility for loop constructs. This results in two possible constructs: a sequence of instructions, and if/then/else clauses. The IPFW filtering language was designed with this in mind. The general form of the language is:

```
condition {
    true action
} else {
    false action
}
```

The false action is optional and typically omitted in normal filter programs. Note that "if" is implied. Initially "else" was also implied, however, this reduced readability so it was added back into the language.

In addition to this generic construct, there is a block statement, which is essentially a series of "if" and "else if" statements. There is also a "case" statement which is similar, but not identical, to a C "switch" statement.

Most actions are either another construct or a terminating condition, such as "accept" or "reject."

## 9 End-User's Perspective

From the end-user's perspective, creating a packet filter involves writing a text file that contains the filter, compiling the filter from the command line, and loading the compiled filter into the kernel from the command line. A user could create the following sample filter in a file called `forward`:

```
#define SERVER   192.168.1.10
#define MAILHOST 192.168.1.15

switch ipprotocol {
case tcp:
    // TCP packets should never come in
    // as fragments
    ipfrag {
        reject;
    }
    // TCP packets need at least 20 bytes
    iplen (<20) {
        reject;
    }
    // We just accept established
    // connections
    established {
        accept;
    }
    // Allow incoming services to
    // some computers
    switch dstaddr {
        case SERVER:
            dstport(ssh/tcp, telnet/tcp,
                    ftp/tcp, http/tcp) {
                accept;
            }
            break;
        case MAILHOST:
            dstport(smtp/tcp) {
                accept;
            }
            break;
    }
    // All other requests are rejected
    // and logged to the kernel socket
    reject[120];
    break;

case udp:
    // Accept non-first fragments
    ipfrag && !ipfirstfrag {
        // But don't allow fragmented
        // UDP headers
        ipoffset(<8) {
            reject;
        }
        accept;
    }
    // UDP packets need at least 8 bytes
    iplen(<8) {
        reject;
    }
    // We just accept all UDP packets
    accept;
    break;

case icmp:
    // We just accept all ICMP packets
    accept;
    break;

default:
    // We reject any other protocols
    // and log them to the socket
    reject[120];
}
```

The user could then compile and load the filter on the forward location in the kernel:

```
# ipfwcmp -o /tmp/ipfw.forward forward
# ipfw forward -tag fwd-filt \\
       -push /tmp/ipfw.forward
```

If the user wanted to examine the effectiveness of their filter program, they could:

```
# ipfw forward -stats
forward filter statistics:
   3068169 packets rejected
           3033113 reported
  19496389 packets accepted
                 0 reported
        14 errors while reporting
         0 unknown disposition
```

## 10 IPFW Specialty Filters

The fact that IPFW is a general filtering framework allows very specialized filters, written in C, to be linked into the kernel. Some examples of this are NAT, IP flow monitoring, and rate filtering. Since IPFW filters have the ability to call other filters, it is possible to use an IPFW filter to do the bulk of the work, but still use a fast C-based hashed lookup scheme on a large pool of addresses.

Rate filtering provides a mechanism that controls how quickly packets are allowed to leave a computer. Different classes of packets can be assigned different rates. Each class of packets is determined by an IPFW classification filter. For example, it is easy to create a class for all outbound http traffic and assign a particular bandwidth limit to it. Additional rate classes could be defined for other protocols and different bandwidth limits applied to each class.

Protocol rate filtering is used in conjunction with a modified circuit-cache to impose a rate limit on individual remote hosts, rather than on a class of packets leaving a computer. For example, a DNS server may limit the number of requests that a particular client can make in a given time period.

The NAT filter provides IP address and port translation services for TCP and UDP traffic. This transparent filtering provides the usual benefits of network address translation.

Flow monitoring gathers data on TCP and UDP traffic between two computers. A TCP flow is a TCP session, while a UDP flow is a series of UDP packets that share the same source and destination address and port. The flow monitoring facility provides data similar to Cisco's NetFlow implementation. This data is useful for network capacity planning as well as high level network protocol analysis.

## 11 Real World Examples

IPFW has many potential uses for anyone who has IP connectivity. Given the multitude of potential filtering operations available, it is instructive to see how IPFW is used by three sample installations.

### 11.1 Home User

The first sample installation uses a small set of the IPFW capabilities. This installation has two network connections, one via a dialup modem using PPP, and a second connection via an IDSL modem. The BSD/OS computer running as a router uses the IPFW capabilities to perform three distinct tasks.

The first is the next-hop routing of outgoing traffic to select between the IDSL and PPP outgoing interfaces, based on the source address of the packet. The second is packet filtering of inbound traffic to the servers located behind the filtering computer. The last is to provide NAT services for other client computers behind the filtering computer.

### 11.2 Shared Corporate Network

The second sample installation uses a larger set of the IPFW capabilities. This installation has only one upstream connection, but multiple different client networks behind the filtering host. The filtering host is also used as a filtering gateway between the different client networks.

The filtering host has a forward filter that allows inbound access for a select number of protocols, to a rigidly defined list of servers on the different client networks. This filter is rather complicated, as it must treat each of the client networks attached to the filtering computer with a different set of filtering options, specified for each of the clients.

The filtering host has a pre-input filter that terminates outbound http requests on the filtering computer, so that a transparent http cache can operate for all the different client networks behind this host. There is also an input filter in use to prevent external users from connecting to the http cache. The input location filtering is done to prevent any inbound TCP connections from ever being established directly to the cache daemon.

The filtering host also has a set of rate filters, to limit how much bandwidth certain network protocols (for example, nntp) are allowed to use at any given time. The rate filtering capability has proven very useful for simulating low-bandwidth links for testing during debugging of network tools that must transfer large quantities of streaming information over long-haul networks. Some of the server computers behind the filtering host also use rate filters to limit the amount of outbound http traffic that may be sent. This is used to enforce bandwidth limits to which the clients have agreed.

The filtering host also has a set of filters in place to filter connections from the "nimda" worm first noted in autumn 2001. This worm attacks web servers and attempts 14 different accesses, probing for known security holes. While none of the servers at this location are vulnerable to this worm, the attacks still cause a great number of extraneous log entries to be made on each web server. There are so many log entries from the attack probes that the normal log entries are drowned out in the noise of the attack entries.

Three filters are used to combat this worm. The first is the system provided, custom filter called "rewrite," which allows sending `RST` packets to one end of a TCP connection. The second filter is a custom, hand-coded BPF assembler filter that is installed into the filter pool with the name "src_dst_swap." This filter swaps the source and destination IP addresses in a packet, as well as the source and destination port addresses. The final filter is an IPFW filter that examines the interior of packets destined for the protected clients' web servers. If the filter detects the "nimda" attack signature after the TCP session has gone through the three-way handshake, the filter calls the "src_dst_swap" filter and then calls the "rewrite" filter. The "rewrite" filter sends a TCP `RST` packet to the internal web server, which will cause it to tear down the just established TCP session. By resetting the TCP connection on only the protected client web server, but not that of the attacking computer, the client web server's kernel resources are immediately freed for reuse. The attacker's kernel resources are intentionally left occupied. By resetting the TCP connection before any data is sent to the web server, no error message is logged by web server. This neatly solves the problem of the extraneous log messages caused by the "nimda" worm. It could be trivially enhanced to deal with other such attacks in the future.

### 11.3 Corporate Firewall

The last sample installation of the IPFW system is a special purpose corporate firewall. A very successful firewall has been built around the IPFW system to dynamically protect a group of servers from malicious dialup users. It uses the IPFW kernel socket and a simple filter at the forward location to read all TCP `SYN` packets coming from a list of CIDR blocks that represent the dialup modem pools. A continually updated database of IP addresses assigned to legitimate dialup users is queried to determine if the `SYN` packet should be allowed. If the packet is permitted, a copy of the packet is sent out a different interface, to the servers to allow establishment of the client's TCP session. Other non-TCP packet filtering is also done on these firewalls using regular packet matching and filtering.

## 12 Future Enhancements to IPFW

There are always ample possibilities for expanding useful software systems, and IPFW is no exception. Under consideration are structural additions as well as changes to make IPFW more accessible to a larger number of people.

Allow loadable kernel modules to implement "Custom Coded Filters." This would allow for the speed of a native implementation of a complex filter, while preserving the ability to reconfigure the filtering system on the fly.

Implement an in-kernel filter compiler that takes BPF programs as input and generates a native (non-interpreted) version of the downloaded BPF filter for execution.

On-demand logging would make it easier to debug filters. This would also allow the assessment of the cost and benefits of making certain system changes.

IPFW support needs to be added back into the KAME IPv6 protocol stack. The IPv6 support needs to be completed, so that next-hop routing can be used with IPv6 addresses.

Expansion of the protocol throttling support. This would allow limiting the number of responses for a particular protocol and provide another method for preventing denial of service attacks.

While the programmable nature of IPFW makes it extremely flexible, programming is not a strength of all system administrators. This lack of programming experience means that some filters are inefficient and others do not take full advantage of IPFW features. A graphical user interface could address these issues.

## 13 Conclusions

The IPFW framework has proven to be extremely flexible. It has been used for purposes never dreamed of in the original design. This is often the hallmark of a good basic design.

IPFW has a complete framework for packet filtering services. It has been extended several times since the original implementation but has never needed to be completely redesigned.

IPFW is very reliable. It has been deployed in applications that have passed terabytes of information between reboots. Machines executing IPFW filters have uptimes in excess of one year, even after multiple filter changes and updates during that time period.

Appropriate documentation is crucial to wide acceptance of a new technology. While the IPFW system has been available for several years and has many powerful and useful features, its acceptance has been slow because of incomplete and opaque documentation.

## 14 Availability

Additional documentation, as well as some of the filters described in this paper, is available at `http://www.pix.net/software/ipfw/`.

## 15 Acknowledgments

The contributions of several people are hereby acknowledged in their efforts to make this a better paper. Jack Flory and Mike Karels for numerous conversations about packet filtering prior to the first line of the IPFW code being written. Bill Cheswick for prompting the need for the circuit cache, which led to the ability to chain filters and the ability to call other filters from a filter. Dave MacKenzie, Josh Osborne, and Chris Ross, for reading draft copies and making useful suggestions. Donn Seely, for guiding this paper through the submission and review process. The staff at the USENIX Association, for they make this conference possible.

## Notes

[1] Even though FreeBSD's ipfirewall is often referred to as `ipfw` it is unrelated to the BSD/OS IPFW system, except in name.

[2] In order to change the filter, one would have to recompile the kernel and reboot the computer with the new kernel.

[3] When work on coding IPFW started, the author searched for the BPF assembler that was described in the original BPF paper. After learning that no actual assembler was ever written, a standalone BPF assembler was written for IPFW.

[4] A language similar to the IPFW filtering language was independently developed for the Ascend GRF router. While the GRF used BSD/OS as the basis of its embedded operating system, the GRF filtering system was independently developed.

## References

[BPF] McCanne, Steve, and Van Jacobson, *"The BSD Packet Filter: A New Architecture for User-level Packet Capture,"* Proceedings of Winter 1993 USENIX Annual Technical Conference. (January, 1993).

[CSPF] Mogul, J.C., and R.F. Rashid, and M.J. Accetta. *"The packet filter: An efficient mechanism for user-level network code."* 11th ACM Symposium on Operating System Principles, November, 1987.

[FreeBSD] The FreeBSD Project. *ipfirewall(4); FreeBSD 4.4-stable Manual Page.* http://www.freebsd.org, June, 1997.

[fwtk] Trusted Information Systems, Inc. *TIS Internet Firewall Toolkit* ftp://ftp.tislabs.com/pub/firewalls/toolkit/, March, 1997.

[ipchains] Russell, Rusty. http://netfilter.samba.org/ipchains/, October, 2000.

[ipfilter] Reed, Darren. http:///www.ipfilter.org/, November, 2001.

[ipfilterhowto] Conoboy, Brendan and Erik Fichtner. *IP Filter Based Firewalls HOWTO* http://www.obfuscation.org/ipf/ipf-howto.pdf, July, 2001.

[iptables] Russell, Rusty. http://netfilter.samba.org/, August, 2001.

[NIT] Sun Microsystems, Inc. *NIT(4P); SunOS 4.1.1 Reference Manual.* Mountain View, California, October, 1990. Part Number 800-5480-10.

[OpenBSD] The OpenBSD Project. *pf(4); OpenBSD 3.0 Manual Page.* http://www.openbsd.org, July, 2001.

[RFC1013] Scheifler, Robert W. *X Window System Protocol, Version 11.* Cambridge, Massachusetts, June, 1987.

[screend] Mogul, Jeffrey C. *"Using screend to implement IP/TCP security policies."* Technical Note TN-2, Digital Equipment Corporation Network Systems Lab, 1991.