NAME

ipfw, **dnctl** - User interface for firewall, traffic shaper, packet scheduler, in-kernel NAT.

SYNOPSIS

FIREWALL CONFIGURATION

ipfw [**-cq**] **add** *rule*
ipfw [**-acdefnNStT**] [**set** *N*] {**list** | **show**} [*rule* | *first-last ...*]
ipfw [**-f** | **-q**] [**set** *N*] **flush**
ipfw [**-q**] [**set** *N*] {**delete** | **zero** | **resetlog**} [*number ...*]

ipfw set [**disable** *number ...*] [**enable** *number ...*]
ipfw set move [**rule**] *number* **to** *number*
ipfw set swap *number number*
ipfw set show

SYSCTL SHORTCUTS

ipfw enable {**firewall** | **altq** | **one_pass** | **debug** | **verbose** | **dyn_keepalive**}
ipfw disable {**firewall** | **altq** | **one_pass** | **debug** | **verbose** | **dyn_keepalive**}

LOOKUP TABLES

ipfw [**set** *N*] **table** *name* **create** *create-options*
ipfw [**set** *N*] **table** {*name* | *all*} **destroy**
ipfw [**set** *N*] **table** *name* **modify** *modify-options*
ipfw [**set** *N*] **table** *name* **swap** *name*
ipfw [**set** *N*] **table** *name* **add** *table-key* [*value*]
ipfw [**set** *N*] **table** *name* **add** [*table-key value ...*]
ipfw [**set** *N*] **table** *name* **atomic add** [*table-key value ...*]
ipfw [**set** *N*] **table** *name* **delete** [*table-key ...*]
ipfw [**set** *N*] **table** *name* **lookup** *addr*
ipfw [**set** *N*] **table** *name* **lock**
ipfw [**set** *N*] **table** *name* **unlock**
ipfw [**set** *N*] **table** {*name* | *all*} **list**
ipfw [**set** *N*] **table** {*name* | *all*} **info**
ipfw [**set** *N*] **table** {*name* | *all*} **detail**
ipfw [**set** *N*] **table** {*name* | *all*} **flush**

DUMMYNET CONFIGURATION (TRAFFIC SHAPER AND PACKET SCHEDULER)

dnctl {**pipe** | **queue** | **sched**} *number* **config** *config-options*
dnctl [**-s** [*field*]] {**pipe** | **queue** | **sched**} {**delete** | **list** | **show**} [*number ...*]

IN-KERNEL NAT

ipfw [**-q**] **nat** *number* **config** *config-options*
ipfw nat *number* **show** {**config** | **log**}

STATEFUL IPv6/IPv4 NETWORK ADDRESS AND PROTOCOL TRANSLATION

ipfw [**set** *N*] **nat64lsn** *name* **create** *create-options*
ipfw [**set** *N*] **nat64lsn** *name* **config** *config-options*
ipfw [**set** *N*] **nat64lsn** {*name* | *all*} {**list** | **show**} [**states**]
ipfw [**set** *N*] **nat64lsn** {*name* | *all*} **destroy**
ipfw [**set** *N*] **nat64lsn** *name* **stats** [**reset**]

STATELESS IPv6/IPv4 NETWORK ADDRESS AND PROTOCOL TRANSLATION

ipfw [**set** *N*] **nat64stl** *name* **create** *create-options*
ipfw [**set** *N*] **nat64stl** *name* **config** *config-options*
ipfw [**set** *N*] **nat64stl** {*name* | *all*} {**list** | **show**}
ipfw [**set** *N*] **nat64stl** {*name* | *all*} **destroy**
ipfw [**set** *N*] **nat64stl** *name* **stats** [**reset**]

XLAT464 CLAT IPv6/IPv4 NETWORK ADDRESS AND PROTOCOL TRANSLATION

ipfw [**set** *N*] **nat64clat** *name* **create** *create-options*
ipfw [**set** *N*] **nat64clat** *name* **config** *config-options*
ipfw [**set** *N*] **nat64clat** {*name* | *all*} {**list** | **show**}
ipfw [**set** *N*] **nat64clat** {*name* | *all*} **destroy**
ipfw [**set** *N*] **nat64clat** *name* **stats** [**reset**]

IPv6-to-IPv6 NETWORK PREFIX TRANSLATION

ipfw [**set** *N*] **nptv6** *name* **create** *create-options*
ipfw [**set** *N*] **nptv6** {*name* | *all*} {**list** | **show**}
ipfw [**set** *N*] **nptv6** {*name* | *all*} **destroy**
ipfw [**set** *N*] **nptv6** *name* **stats** [**reset**]

INTERNAL DIAGNOSTICS

ipfw internal iflist
ipfw internal talist
ipfw internal vlist

LIST OF RULES AND PREPROCESSING

ipfw [**-cfnNqS**] [**-p** *preproc* [*preproc-flags*]] *pathname*

DESCRIPTION

The **ipfw** utility is the user interface for controlling the ipfw(4) firewall, the dummynet(4) traffic shaper/packet scheduler, and the in-kernel NAT services.

A firewall configuration, or *ruleset*, is made of a list of *rules* numbered from 1 to 65535. Packets are passed to the firewall from a number of different places in the protocol stack (depending on the source and destination of the packet, it is possible for the firewall to be invoked multiple times on the same packet). The packet passed to the firewall is compared against each of the rules in the *ruleset*, in rule-number order (multiple rules with the same number are permitted, in which case they are processed in order of insertion). When a match is found, the action corresponding to the matching rule is performed.

Depending on the action and certain system settings, packets can be reinjected into the firewall at some rule after the matching one for further processing.

A ruleset always includes a *default* rule (numbered 65535) which cannot be modified or deleted, and matches all packets. The action associated with the *default* rule can be either **deny** or **allow** depending on how the kernel is configured.

If the ruleset includes one or more rules with the **keep-state**, **record-state**, **limit** or **set-limit** option, the firewall will have a *stateful* behaviour, i.e., upon a match it will create *dynamic rules*, i.e., rules that match packets with the same 5-tuple (protocol, source and destination addresses and ports) as the packet which caused their creation. Dynamic rules, which have a limited lifetime, are checked at the first occurrence of a **check-state**, **keep-state** or **limit** rule, and are typically used to open the firewall on-demand to legitimate traffic only. Please note, that **keep-state** and **limit** imply implicit **check-state** for all packets (not only these matched by the rule) but **record-state** and **set-limit** have no implicit **check-state**. See the *STATEFUL FIREWALL* and *EXAMPLES* Sections below for more information on the stateful behaviour of **ipfw**.

All rules (including dynamic ones) have a few associated counters: a packet count, a byte count, a log count and a timestamp indicating the time of the last match. Counters can be displayed or reset with **ipfw** commands.

Each rule belongs to one of 32 different *sets* , and there are **ipfw** commands to atomically manipulate sets, such as enable, disable, swap sets, move all rules in a set to another one, delete all rules in a set. These can be useful to install temporary configurations, or to test them. See Section *SETS OF RULES* for more information on *sets*.

Rules can be added with the **add** command; deleted individually or in groups with the **delete** command, and globally (except those in set 31) with the **flush** command; displayed, optionally with the content of the counters, using the **show** and **list** commands. Finally, counters can be reset with the **zero** and **resetlog** commands.

COMMAND OPTIONS

The following general options are available when invoking **ipfw**:

-a Show counter values when listing rules. The **show** command implies this option.

-b Only show the action and the comment, not the body of a rule. Implies **-c**.

-c When entering or showing rules, print them in compact form, i.e., omitting the "ip from any to any" string when this does not carry any additional information.

-d When listing, show dynamic rules in addition to static ones.

-D When listing, show only dynamic states. When deleting, delete only dynamic states.

-f Run without prompting for confirmation for commands that can cause problems if misused, i.e., **flush**. If there is no tty associated with the process, this is implied. The **delete** command with this flag ignores possible errors, i.e., nonexistent rule number. And for batched commands execution continues with the next command.

-i When listing a table (see the *LOOKUP TABLES* section below for more information on lookup tables), format values as IP addresses. By default, values are shown as integers.

-n Only check syntax of the command strings, without actually passing them to the kernel.

-N Try to resolve addresses and service names in output.

-q Be quiet when executing the **add**, **nat**, **zero**, **resetlog** or **flush** commands; (implies **-f**). This is useful when updating rulesets by executing multiple **ipfw** commands in a script (e.g., 'sh /etc/rc.firewall'), or by processing a file with many **ipfw** rules across a remote login session. It also stops a table add or delete from failing if the entry already exists or is not present.

The reason why this option may be important is that for some of these actions, **ipfw** may print a message; if the action results in blocking the traffic to the remote client, the remote login session will be closed and the rest of the ruleset will not be processed. Access to the console would then be required to recover.

-S When listing rules, show the *set* each rule belongs to. If this flag is not specified, disabled rules will not be listed.

-s [*field*]

When listing pipes, sort according to one of the four counters (total or current packets or bytes).

-t When listing, show last match timestamp converted with **ctime**().

-T When listing, show last match timestamp as seconds from the epoch. This form can be more convenient for postprocessing by scripts.

LIST OF RULES AND PREPROCESSING

To ease configuration, rules can be put into a file which is processed using **ipfw** as shown in the last synopsis line. An absolute *pathname* must be used. The file will be read line by line and applied as arguments to the **ipfw** utility.

Optionally, a preprocessor can be specified using **-p** *preproc* where *pathname* is to be piped through. Useful preprocessors include cpp(1) and m4(1). If *preproc* does not start with a slash ('/') as its first character, the usual PATH name search is performed. Care should be taken with this in environments where not all file systems are mounted (yet) by the time **ipfw** is being run (e.g. when they are mounted over NFS). Once **-p** has been specified, any additional arguments are passed on to the preprocessor for interpretation. This allows for flexible configuration files (like conditionalizing them on the local hostname) and the use of macros to centralize frequently required arguments like IP addresses.

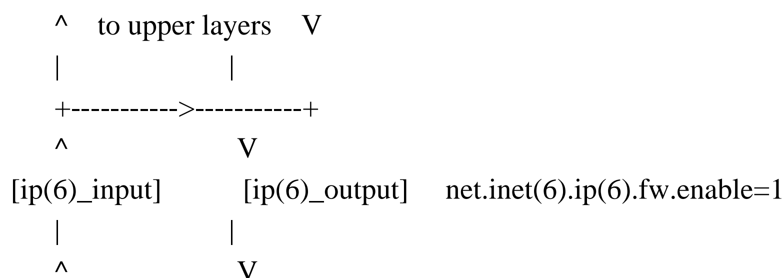
TRAFFIC SHAPER CONFIGURATION

The **dnctl pipe**, **queue** and **sched** commands are used to configure the traffic shaper and packet scheduler. See the *TRAFFIC SHAPER (DUMMYNET) CONFIGURATION* Section below for details.

If the world and the kernel get out of sync the **ipfw** ABI may break, preventing you from being able to add any rules. This can adversely affect the booting process. You can use **ipfw disable firewall** to temporarily disable the firewall to regain access to the network, allowing you to fix the problem.

PACKET FLOW

A packet is checked against the active ruleset in multiple places in the protocol stack, under control of several sysctl variables. These places and variables are shown below, and it is important to have this picture in mind in order to design a correct ruleset.



```
     ^    to upper layers    V
     |                       |
     +----------->-----------+
     ^                       V
[ip(6)_input]          [ip(6)_output]     net.inet(6).ip(6).fw.enable=1
     |                       |
     ^                       V
```

```
    [ether_demux]       [ether_output_frame]  net.link.ether.ipfw=1
         |                    |
         +-->--[bdg_forward]-->--+          net.link.bridge.ipfw=1
         ^                    V
         |     to devices       |
```

The number of times the same packet goes through the firewall can vary between 0 and 4 depending on packet source and destination, and system configuration.

Note that as packets flow through the stack, headers can be stripped or added to it, and so they may or may not be available for inspection. E.g., incoming packets will include the MAC header when **ipfw** is invoked from **ether_demux()**, but the same packets will have the MAC header stripped off when **ipfw** is invoked from **ip_input()** or **ip6_input()**.

Also note that each packet is always checked against the complete ruleset, irrespective of the place where the check occurs, or the source of the packet. If a rule contains some match patterns or actions which are not valid for the place of invocation (e.g. trying to match a MAC header within **ip_input** or **ip6_input),** the match pattern will not match, but a **not** operator in front of such patterns *will* cause the pattern to *always* match on those packets. It is thus the responsibility of the programmer, if necessary, to write a suitable ruleset to differentiate among the possible places. **skipto** rules can be useful here, as an example:

```
# packets from ether_demux or bdg_forward
ipfw add 10 skipto 1000 all from any to any layer2 in
# packets from ip_input
ipfw add 10 skipto 2000 all from any to any not layer2 in
# packets from ip_output
ipfw add 10 skipto 3000 all from any to any not layer2 out
# packets from ether_output_frame
ipfw add 10 skipto 4000 all from any to any layer2 out
```

(yes, at the moment there is no way to differentiate between ether_demux and bdg_forward).

Also note that only actions **allow**, **deny**, **netgraph**, **ngtee** and related to **dummynet** are processed for **layer2** frames and all other actions act as if they were **allow** for such frames. Full set of actions is supported for IP packets without **layer2** headers only. For example, **divert** action does not divert **layer2** frames.

SYNTAX

In general, each keyword or argument must be provided as a separate command line argument, with no

leading or trailing spaces. Keywords are case-sensitive, whereas arguments may or may not be case-sensitive depending on their nature (e.g. uid's are, hostnames are not).

Some arguments (e.g., port or address lists) are comma-separated lists of values. In this case, spaces after commas ',' are allowed to make the line more readable. You can also put the entire command (including flags) into a single argument. E.g., the following forms are equivalent:

```
ipfw -q add deny src-ip 10.0.0.0/24,127.0.0.1/8
ipfw -q add deny src-ip 10.0.0.0/24, 127.0.0.1/8
ipfw "-q add deny src-ip 10.0.0.0/24, 127.0.0.1/8"
```

RULE FORMAT

The format of firewall rules is the following:

[*rule_number*] [**set** *set_number*] [**prob** *match_probability*] *action* [**log** [**logamount** *number*]]
[**altq** *queue*] [{**tag** | **untag**} *number*] *body*

where the body of the rule specifies which information is used for filtering packets, among the following:

Layer2 header fields	When available
IPv4 and IPv6 Protocol	SCTP, TCP, UDP, ICMP, etc.
Source and dest. addresses and ports	
Direction	See Section *PACKET FLOW*
Transmit and receive interface	By name or address
Misc. IP header fields	Version, type of service, datagram length, identification, fragmentation flags, Time To Live
IP options	
IPv6 Extension headers	Fragmentation, Hop-by-Hop options, Routing Headers, Source routing rthdr0, Mobile IPv6 rthdr2, IPSec options.
IPv6 Flow-ID	
Misc. TCP header fields	TCP flags (SYN, FIN, ACK, RST, etc.), sequence number, acknowledgment number, window
TCP options	
ICMP types	for ICMP packets
ICMP6 types	for ICMP6 packets
User/group ID	When the packet can be associated with a local socket.
Divert status	Whether a packet came from a divert socket (e.g., natd(8)).
Fib annotation state	Whether a packet has been tagged for using a specific FIB (routing table) in future forwarding decisions.

Note that some of the above information, e.g. source MAC or IP addresses and TCP/UDP ports, can be easily spoofed, so filtering on those fields alone might not guarantee the desired results.

rule_number

Each rule is associated with a *rule_number* in the range 1..65535, with the latter reserved for the *default* rule. Rules are checked sequentially by rule number. Multiple rules can have the same number, in which case they are checked (and listed) according to the order in which they have been added. If a rule is entered without specifying a number, the kernel will assign one in such a way that the rule becomes the last one before the *default* rule. Automatic rule numbers are assigned by incrementing the last non-default rule number by the value of the sysctl variable *net.inet.ip.fw.autoinc_step* which defaults to 100. If this is not possible (e.g. because we would go beyond the maximum allowed rule number), the number of the last non-default value is used instead.

set *set_number*

Each rule is associated with a *set_number* in the range 0..31. Sets can be individually disabled and enabled, so this parameter is of fundamental importance for atomic ruleset manipulation. It can be also used to simplify deletion of groups of rules. If a rule is entered without specifying a set number, set 0 will be used.
Set 31 is special in that it cannot be disabled, and rules in set 31 are not deleted by the **ipfw flush** command (but you can delete them with the **ipfw delete set 31** command). Set 31 is also used for the *default* rule.

prob *match_probability*

A match is only declared with the specified probability (floating point number between 0 and 1). This can be useful for a number of applications such as random packet drop or (in conjunction with **dummynet**) to simulate the effect of multiple paths leading to out-of-order packet delivery.

Note: this condition is checked before any other condition, including ones such as **keep-state** or **check-state** which might have side effects.

log [**logamount** *number*]

Packets matching a rule with the **log** keyword will be made available for logging in two ways: if the sysctl variable *net.inet.ip.fw.verbose* is set to 0 (default), one can use bpf(4) attached to the ipfw0 pseudo interface. This pseudo interface can be created manually after a system boot by using the following command:

```
# ifconfig ipfw0 create
```

Or, automatically at boot time by adding the following line to the rc.conf(5) file:

firewall_logif="YES"

There is zero overhead when no bpf(4) is attached to the pseudo interface.

If *net.inet.ip.fw.verbose* is set to 1, packets will be logged to syslogd(8) with a LOG_SECURITY facility up to a maximum of **logamount** packets. If no **logamount** is specified, the limit is taken from the sysctl variable *net.inet.ip.fw.verbose_limit*. In both cases, a value of 0 means unlimited logging.

Once the limit is reached, logging can be re-enabled by clearing the logging counter or the packet counter for that entry, see the **resetlog** command.

Note: logging is done after all other packet matching conditions have been successfully verified, and before performing the final action (accept, deny, etc.) on the packet.

tag *number*

When a packet matches a rule with the **tag** keyword, the numeric tag for the given *number* in the range 1..65534 will be attached to the packet. The tag acts as an internal marker (it is not sent out over the wire) that can be used to identify these packets later on. This can be used, for example, to provide trust between interfaces and to start doing policy-based filtering. A packet can have multiple tags at the same time. Tags are "sticky", meaning once a tag is applied to a packet by a matching rule it exists until explicit removal. Tags are kept with the packet everywhere within the kernel, but are lost when the packet leaves the kernel, for example, on transmitting packet out to the network or sending packet to a divert(4) socket.

To check for previously applied tags, use the **tagged** rule option. To delete previously applied tag, use the **untag** keyword.

Note: since tags are kept with the packet everywhere in kernelspace, they can be set and unset anywhere in the kernel network subsystem (using the mbuf_tags(9) facility), not only by means of the ipfw(4) **tag** and **untag** keywords. For example, there can be a specialized netgraph(4) node doing traffic analyzing and tagging for later inspecting in firewall.

untag *number*

When a packet matches a rule with the **untag** keyword, the tag with the number *number* is searched among the tags attached to this packet and, if found, removed from it. Other tags bound to packet, if present, are left untouched.

setmark *value* | *tablearg*

When a packet matches a rule with the **setmark** keyword, a 32-bit numeric mark is assigned to

the packet. The mark is an extension to the tags. As tags, mark is "sticky" so the value is kept the same within the kernel and is lost when the packet leaves the kernel. Unlike tags, mark can be matched as a lookup table key or compared with bitwise mask applied against another value. Each packet can have only one mark, so **setmark** always overwrites the previous mark value.

The initial mark value is 0. To check the current mark value, use the **mark** rule option. Mark *value* can be entered as decimal or hexadecimal (if prefixed by 0x), and they are always printed as hexadecimal.

altq *queue*

When a packet matches a rule with the **altq** keyword, the ALTQ identifier for the given *queue* (see altq(4)) will be attached. Note that this ALTQ tag is only meaningful for packets going "out" of IPFW, and not being rejected or going to divert sockets. Note that if there is insufficient memory at the time the packet is processed, it will not be tagged, so it is wise to make your ALTQ "default" queue policy account for this. If multiple **altq** rules match a single packet, only the first one adds the ALTQ classification tag. In doing so, traffic may be shaped by using **count altq** *queue* rules for classification early in the ruleset, then later applying the filtering decision. For example, **check-state** and **keep-state** rules may come later and provide the actual filtering decisions in addition to the fallback ALTQ tag.

You must run pfctl(8) to set up the queues before IPFW will be able to look them up by name, and if the ALTQ disciplines are rearranged, the rules in containing the queue identifiers in the kernel will likely have gone stale and need to be reloaded. Stale queue identifiers will probably result in misclassification.

All system ALTQ processing can be turned on or off via **ipfw enable** *altq* and **ipfw disable** *altq*. The usage of *net.inet.ip.fw.one_pass* is irrelevant to ALTQ traffic shaping, as the actual rule action is followed always after adding an ALTQ tag.

RULE ACTIONS

A rule can be associated with one of the following actions, which will be executed when the packet matches the body of the rule.

allow | **accept** | **pass** | **permit**

Allow packets that match rule. The search terminates.

check-state [*:flowname* | **:any**]

Checks the packet against the dynamic ruleset. If a match is found, execute the action associated with the rule which generated this dynamic rule, otherwise move to the next rule.
Check-state rules do not have a body. If no **check-state** rule is found, the dynamic ruleset is

checked at the first **keep-state** or **limit** rule. The *:flowname* is symbolic name assigned to dynamic rule by **keep-state** opcode. The special flowname **:any** can be used to ignore states flowname when matching. The **:default** keyword is special name used for compatibility with old rulesets.

count Update counters for all packets that match rule. The search continues with the next rule.

deny | **drop**
Discard packets that match this rule. The search terminates.

divert *port*
Divert packets that match this rule to the divert(4) socket bound to port *port*. The search terminates.

fwd | **forward** *ipaddr* | *tablearg*[,*port*]
Change the next-hop on matching packets to *ipaddr*, which can be an IP address or a host name. The next hop can also be supplied by the last table looked up for the packet by using the **tablearg** keyword instead of an explicit address. The search terminates if this rule matches.

If *ipaddr* is a local address, then matching packets will be forwarded to *port* (or the port number in the packet if one is not specified in the rule) on the local machine.
If *ipaddr* is not a local address, then the port number (if specified) is ignored, and the packet will be forwarded to the remote address, using the route as found in the local routing table for that IP.
A *fwd* rule will not match layer2 packets (those received on ether_input, ether_output, or bridged).
The **fwd** action does not change the contents of the packet at all. In particular, the destination address remains unmodified, so packets forwarded to another system will usually be rejected by that system unless there is a matching rule on that system to capture them. For packets forwarded locally, the local address of the socket will be set to the original destination address of the packet. This makes the netstat(1) entry look rather weird but is intended for use with transparent proxy servers.

nat *nat_nr* | *global* | *tablearg*
Pass packet to a nat instance (for network address translation, address redirect, etc.): see the *NETWORK ADDRESS TRANSLATION (NAT)* Section for further information.

nat64lsn *name*
Pass packet to a stateful NAT64 instance (for IPv6/IPv4 network address and protocol translation): see the *IPv6/IPv4 NETWORK ADDRESS AND PROTOCOL TRANSLATION* Section for further information.

nat64stl *name*
Pass packet to a stateless NAT64 instance (for IPv6/IPv4 network address and protocol translation): see the *IPv6/IPv4 NETWORK ADDRESS AND PROTOCOL TRANSLATION* Section for further information.

nat64clat *name*
Pass packet to a CLAT NAT64 instance (for client-side IPv6/IPv4 network address and protocol translation): see the *IPv6/IPv4 NETWORK ADDRESS AND PROTOCOL TRANSLATION* Section for further information.

nptv6 *name*
Pass packet to a NPTv6 instance (for IPv6-to-IPv6 network prefix translation): see the *IPv6-to-IPv6 NETWORK PREFIX TRANSLATION (NPTv6)* Section for further information.

pipe *pipe_nr*
Pass packet to a **dummynet** "pipe" (for bandwidth limitation, delay, etc.). See the *TRAFFIC SHAPER (DUMMYNET) CONFIGURATION* Section for further information. The search terminates; however, on exit from the pipe and if the sysctl(8) variable *net.inet.ip.fw.one_pass* is not set, the packet is passed again to the firewall code starting from the next rule.

queue *queue_nr*
Pass packet to a **dummynet** "queue" (for bandwidth limitation using WF2Q+).

reject (Deprecated). Synonym for **unreach host**.

reset Discard packets that match this rule, and if the packet is a TCP packet, try to send a TCP reset (RST) notice. The search terminates.

reset6 Discard packets that match this rule, and if the packet is a TCP packet, try to send a TCP reset (RST) notice. The search terminates.

skipto *number* | *tablearg*
Skip all subsequent rules numbered less than *number*. The search continues with the first rule numbered *number* or higher. It is possible to use the **tablearg** keyword with a skipto for a *computed* skipto. Skipto may work either in O(log(N)) or in O(1) depending on amount of memory and/or sysctl variables. See the *SYSCTL VARIABLES* section for more details.

call *number* | *tablearg*
The current rule number is saved in the internal stack and ruleset processing continues with the first rule numbered *number* or higher. If later a rule with the **return** action is encountered, the

processing returns to the first rule with number of this **call** rule plus one or higher (the same behaviour as with packets returning from divert(4) socket after a **divert** action). This could be used to make somewhat like an assembly language "subroutine" calls to rules with common checks for different interfaces, etc.

Rule with any number could be called, not just forward jumps as with **skipto**. So, to prevent endless loops in case of mistakes, both **call** and **return** actions don't do any jumps and simply go to the next rule if memory cannot be allocated or stack overflowed/underflowed.

Internally stack for rule numbers is implemented using mbuf_tags(9) facility and currently has size of 16 entries. As mbuf tags are lost when packet leaves the kernel, **divert** should not be used in subroutines to avoid endless loops and other undesired effects.

return Takes rule number saved to internal stack by the last **call** action and returns ruleset processing to the first rule with number greater than number of corresponding **call** rule. See description of the **call** action for more details.

Note that **return** rules usually end a "subroutine" and thus are unconditional, but **ipfw** command-line utility currently requires every action except **check-state** to have body. While it is sometimes useful to return only on some packets, usually you want to print just "return" for readability. A workaround for this is to use new syntax and **-c** switch:

```
# Add a rule without actual body
ipfw add 2999 return via any

# List rules without "from any to any" part
ipfw -c list
```

This cosmetic annoyance may be fixed in future releases.

tee *port*

Send a copy of packets matching this rule to the divert(4) socket bound to port *port*. The search continues with the next rule.

unreach *code* [mtu]

Discard packets that match this rule, and try to send an ICMP unreachable notice with code *code*, where *code* is a number from 0 to 255, or one of these aliases: **net**, **host**, **protocol**, **port**, **needfrag**, **srcfail**, **net-unknown**, **host-unknown**, **isolated**, **net-prohib**, **host-prohib**, **tosnet**, **toshost**, **filter-prohib**, **host-precedence** or **precedence-cutoff**. The **needfrag** code may have an optional *mtu* parameter. If specified, the MTU value will be put into generated ICMP packet. The search

terminates.

unreach6 *code*

Discard packets that match this rule, and try to send an ICMPv6 unreachable notice with code *code*, where *code* is a number from 0, 1, 3 or 4, or one of these aliases: **no-route, admin-prohib, address** or **port**. The search terminates.

netgraph *cookie*

Divert packet into netgraph with given *cookie*. The search terminates. If packet is later returned from netgraph it is either accepted or continues with the next rule, depending on *net.inet.ip.fw.one_pass* sysctl variable.

ngtee *cookie*

A copy of packet is diverted into netgraph, original packet continues with the next rule. See ng_ipfw(4) for more information on **netgraph** and **ngtee** actions.

setfib *fibnum* | *tablearg*

The packet is tagged so as to use the FIB (routing table) *fibnum* in any subsequent forwarding decisions. In the current implementation, this is limited to the values 0 through 15, see setfib(2). Processing continues at the next rule. It is possible to use the **tablearg** keyword with setfib. If the tablearg value is not within the compiled range of fibs, the packet's fib is set to 0.

setdscp *DSCP* | *number* | *tablearg*

Set specified DiffServ codepoint for an IPv4/IPv6 packet. Processing continues at the next rule. Supported values are:

cs0 (000000), **cs1** (001000), **cs2** (010000), **cs3** (011000), **cs4** (100000), **cs5** (101000), **cs6** (110000), **cs7** (111000), **af11** (001010), **af12** (001100), **af13** (001110), **af21** (010010), **af22** (010100), **af23** (010110), **af31** (011010), **af32** (011100), **af33** (011110), **af41** (100010), **af42** (100100), **af43** (100110), **va** (101100), **ef** (101110), **be** (000000). Additionally, DSCP value can be specified by number (0..63). It is also possible to use the **tablearg** keyword with setdscp. If the tablearg value is not within the 0..63 range, lower 6 bits of supplied value are used.

tcp-setmss *mss*

Set the Maximum Segment Size (MSS) in the TCP segment to value *mss*. The kernel module **ipfw_pmod** should be loaded or kernel should have **options IPFIREWALL_PMOD** to be able use this action. This command does not change a packet if original MSS value is lower than specified value. Both TCP over IPv4 and over IPv6 are supported. Regardless of matched a packet or not by the **tcp-setmss** rule, the search continues with the next rule.

reass Queue and reassemble IPv4 fragments. If the packet is not fragmented, counters are updated and processing continues with the next rule. If the packet is the last logical fragment, the packet is reassembled and, if *net.inet.ip.fw.one_pass* is set to 0, processing continues with the next rule. Otherwise, the packet is allowed to pass and the search terminates. If the packet is a fragment in the middle of a logical group of fragments, it is consumed and processing stops immediately.

Fragment handling can be tuned via *net.inet.ip.maxfragpackets* and *net.inet.ip.maxfragsperpacket* which limit, respectively, the maximum number of processable fragments (default: 800) and the maximum number of fragments per packet (default: 16).

NOTA BENE: since fragments do not contain port numbers, they should be avoided with the **reass** rule. Alternatively, direction-based (like **in** / **out**) and source-based (like **via**) match patterns can be used to select fragments.

Usually a simple rule like:

```
# reassemble incoming fragments
ipfw add reass all from any to any in
```

is all you need at the beginning of your ruleset.

abort Discard packets that match this rule, and if the packet is an SCTP packet, try to send an SCTP packet containing an ABORT chunk. The search terminates.

abort6
Discard packets that match this rule, and if the packet is an SCTP packet, try to send an SCTP packet containing an ABORT chunk. The search terminates.

RULE BODY

The body of a rule contains zero or more patterns (such as specific source and destination addresses or ports, protocol options, incoming or outgoing interfaces, etc.) that the packet must match in order to be recognised. In general, the patterns are connected by (implicit) **and** operators -- i.e., all must match in order for the rule to match. Individual patterns can be prefixed by the **not** operator to reverse the result of the match, as in

```
ipfw add 100 allow ip from not 1.2.3.4 to any
```

Additionally, sets of alternative match patterns (*or-blocks*) can be constructed by putting the patterns in lists enclosed between parentheses ( ) or braces { }, and using the **or** operator as follows:

ipfw add 100 allow ip from { x or not y or z } to any

Only one level of parentheses is allowed. Beware that most shells have special meanings for parentheses or braces, so it is advisable to put a backslash \ in front of them to prevent such interpretations.

The body of a rule must in general include a source and destination address specifier. The keyword *any* can be used in various places to specify that the content of a required field is irrelevant.

The rule body has the following format:

[*proto* **from** *src* **to** *dst*] [*options*]

The first part (proto from src to dst) is for backward compatibility with earlier versions of FreeBSD. In modern FreeBSD any match pattern (including MAC headers, IP protocols, addresses and ports) can be specified in the *options* section.

Rule fields have the following meaning:

proto: *protocol* | **{** *protocol* **or ... }**

protocol: [**not**] *protocol-name* | *protocol-number*

An IP protocol specified by number or name (for a complete list see */etc/protocols*), or one of the following keywords:

ip4 | **ipv4**

Matches IPv4 packets.

ip6 | **ipv6**

Matches IPv6 packets.

ip | **all** Matches any packet.

The **ipv6** in **proto** option will be treated as inner protocol. And, the **ipv4** is not available in **proto** option.

The **{** *protocol* **or ... }** format (an *or-block*) is provided for convenience only but its use is deprecated.

src and *dst*: {**addr** | **{** *addr* **or ... }**} [[**not**] *ports*]

An address (or a list, see below) optionally followed by *ports* specifiers.

The second format (*or-block* with multiple addresses) is provided for convenience only and its use is discouraged.

addr: [**not**] {**any** | **me** | **me6** | **table**(*name*[,*value*]) | *addr-list* | *addr-set*}

any Matches any IP address.

me Matches any IP address configured on an interface in the system.

me6 Matches any IPv6 address configured on an interface in the system. The address list is evaluated at the time the packet is analysed.

table(*name*[,*value*])
Matches any IPv4 or IPv6 address for which an entry exists in the lookup table *number*. If an optional 32-bit unsigned *value* is also specified, an entry will match only if it has this value. See the *LOOKUP TABLES* section below for more information on lookup tables.

addr-list: *ip-addr*[,*addr-list*]

ip-addr:
A host or subnet address specified in one of the following ways:

numeric-ip | *hostname*
Matches a single IPv4 address, specified as dotted-quad or a hostname. Hostnames are resolved at the time the rule is added to the firewall list.

addr/*masklen*
Matches all addresses with base *addr* (specified as an IP address, a network number, or a hostname) and mask width of **masklen** bits. As an example, 1.2.3.4/25 or 1.2.3.0/25 will match all IP numbers from 1.2.3.0 to 1.2.3.127 .

addr:*mask*
Matches all addresses with base *addr* (specified as an IP address, a network number, or a hostname) and the mask of *mask*, specified as a dotted quad. As an example, 1.2.3.4:255.0.255.0 or 1.0.3.0:255.0.255.0 will match 1.*.3.*. This form is advised only for non-contiguous masks. It is better to resort to the *addr*/*masklen* format for contiguous masks, which is more compact and less error-prone.

addr-set: *addr*[/*masklen*]**{***list***}**

list: {*num* | *num-num*}[,*list*]

Matches all addresses with base address *addr* (specified as an IP address, a network number, or a hostname) and whose last byte is in the list between braces { } . Note that there must be no spaces between braces and numbers (spaces after commas are allowed). Elements of the list can be specified as single entries or ranges. The *masklen* field is used to limit the size of the set of addresses, and can have any value between 24 and 32. If not specified, it will be assumed as 24. This format is particularly useful to handle sparse address sets within a single rule. Because the matching occurs using a bitmask, it takes constant time and dramatically reduces the complexity of rulesets.
As an example, an address specified as 1.2.3.4/24{128,35-55,89} or 1.2.3.0/24{128,35-55,89} will match the following IP addresses:
1.2.3.128, 1.2.3.35 to 1.2.3.55, 1.2.3.89 .

addr6-list: *ip6-addr*[,*addr6-list*]

ip6-addr:

A host or subnet specified one of the following ways:

numeric-ip | *hostname*

Matches a single IPv6 address as allowed by inet_pton(3) or a hostname. Hostnames are resolved at the time the rule is added to the firewall list.

addr/*masklen*

Matches all IPv6 addresses with base *addr* (specified as allowed by inet_pton(3) or a hostname) and mask width of **masklen** bits.

addr/*mask*

Matches all IPv6 addresses with base *addr* (specified as allowed by inet_pton(3) or a hostname) and the mask of *mask*, specified as allowed by inet_pton(3). As an example, fe::640:0:0/ffff::ffff:ffff:0:0 will match fe:*:*:*:0:640:*:*. This form is advised only for non-contiguous masks. It is better to resort to the *addr*/*masklen* format for contiguous masks, which is more compact and less error-prone.

No support for sets of IPv6 addresses is provided because IPv6 addresses are typically random past the initial prefix.

ports: {*port* | *port-port*}[,*ports*]

For protocols which support port numbers (such as SCTP, TCP and UDP), optional **ports** may be

specified as one or more ports or port ranges, separated by commas but no spaces, and an optional **not** operator. The '-' notation specifies a range of ports (including boundaries).

Service names (from */etc/services*) may be used instead of numeric port values. The length of the port list is limited to 30 ports or ranges, though one can specify larger ranges by using an *or-block* in the **options** section of the rule.

A backslash ('\') can be used to escape the dash ('-') character in a service name (from a shell, the backslash must be typed twice to avoid the shell itself interpreting it as an escape character).

```
ipfw add count tcp from any ftp\\-data-ftp to any
```

Fragmented packets which have a non-zero offset (i.e., not the first fragment) will never match a rule which has one or more port specifications. See the **frag** option for details on matching fragmented packets.

RULE OPTIONS (MATCH PATTERNS)

Additional match patterns can be used within rules. Zero or more of these so-called *options* can be present in a rule, optionally prefixed by the **not** operand, and possibly grouped into *or-blocks*.

The following match patterns can be used (listed in alphabetical order):

// this is a comment.
: Inserts the specified text as a comment in the rule. Everything following // is considered as a comment and stored in the rule. You can have comment-only rules, which are listed as having a **count** action followed by the comment.

bridged
: Alias for **layer2**.

defer-immediate-action | defer-action
: A rule with this option will not perform normal action upon a match. This option is intended to be used with **record-state** or **keep-state** as the dynamic rule, created but ignored on match, will work as intended. Rules with both **record-state** and **defer-immediate-action** create a dynamic rule and continue with the next rule without actually performing the action part of this rule. When the rule is later activated via the state table, the action is performed as usual.

diverted
: Matches only packets generated by a divert socket.

diverted-loopback
Matches only packets coming from a divert socket back into the IP stack input for delivery.

diverted-output
Matches only packets going from a divert socket back outward to the IP stack output for delivery.

dst-ip *ip-address*
Matches IPv4 packets whose destination IP is one of the address(es) specified as argument.

{**dst-ip6** | **dst-ipv6**} *ip6-address*
Matches IPv6 packets whose destination IP is one of the address(es) specified as argument.

dst-port *ports*
Matches IP packets whose destination port is one of the port(s) specified as argument.

established
Matches TCP packets that have the RST or ACK bits set.

ext6hdr *header*
Matches IPv6 packets containing the extended header given by *header*. Supported headers are:

Fragment, (**frag**), Hop-to-hop options (**hopopt**), any type of Routing Header (**route**), Source routing Routing Header Type 0 (**rthdr0**), Mobile IPv6 Routing Header Type 2 (**rthdr2**), Destination options (**dstopt**), IPSec authentication headers (**ah**), and IPsec encapsulated security payload headers (**esp**).

fib *fibnum*
Matches a packet that has been tagged to use the given FIB (routing table) number.

flow *table*(*name*[,*value*])
Search for the flow entry in lookup table *name*. If not found, the match fails. Otherwise, the match succeeds and **tablearg** is set to the value extracted from the table.

This option can be useful to quickly dispatch traffic based on certain packet fields. See the *LOOKUP TABLES* section below for more information on lookup tables.

flow-id *labels*
Matches IPv6 packets containing any of the flow labels given in *labels*. *labels* is a comma separated list of numeric flow labels.

dst-mac *table*(*name*[,*value*])

Search for the destination MAC address entry in lookup table *name*. If not found, the match fails. Otherwise, the match succeeds and **tablearg** is set to the value extracted from the table.

src-mac *table*(*name*[,*value*])

Search for the source MAC address entry in lookup table *name*. If not found, the match fails. Otherwise, the match succeeds and **tablearg** is set to the value extracted from the table.

frag *spec*

Matches IPv4 packets whose **ip_off** field contains the comma separated list of IPv4 fragmentation options specified in *spec*. The recognized options are: **df** (don't fragment), **mf** (more fragments), **rf** (reserved fragment bit) **offset** (non-zero fragment offset). The absence of a particular options may be denoted with a '!'.

Empty list of options defaults to matching on non-zero fragment offset. Such rule would match all not the first fragment datagrams, both IPv4 and IPv6. This is a backward compatibility with older rulesets.

gid *group*

Matches all TCP or UDP packets sent by or received for a *group*. A *group* may be specified by name or number.

jail *jail*

Matches all TCP or UDP packets sent by or received for the jail whose ID or name is *jail*.

icmptypes *types*

Matches ICMP packets whose ICMP type is in the list *types*. The list may be specified as any combination of individual types (numeric) separated by commas. *Ranges are not allowed*. The supported ICMP types are:

echo reply (**0**), destination unreachable (**3**), source quench (**4**), redirect (**5**), echo request (**8**), router advertisement (**9**), router solicitation (**10**), time-to-live exceeded (**11**), IP header bad (**12**), timestamp request (**13**), timestamp reply (**14**), information request (**15**), information reply (**16**), address mask request (**17**) and address mask reply (**18**).

icmp6types *types*

Matches ICMP6 packets whose ICMP6 type is in the list of *types*. The list may be specified as any combination of individual types (numeric) separated by commas. *Ranges are not allowed*.

in | **out**

Matches incoming or outgoing packets, respectively. **in** and **out** are mutually exclusive (in fact, **out** is implemented as **not in**).

ipid *id-list*
Matches IPv4 packets whose **ip_id** field has value included in *id-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

iplen *len-list*
Matches IP packets whose total length, including header and data, is in the set *len-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

ipoptions *spec*
Matches packets whose IPv4 header contains the comma separated list of options specified in *spec*. The supported IP options are:

ssrr (strict source route), **lsrr** (loose source route), **rr** (record packet route) and **ts** (timestamp). The absence of a particular option may be denoted with a '!'.

ipprecedence *precedence*
Matches IPv4 packets whose precedence field is equal to *precedence*.

ipsec Matches packets that have IPSEC history associated with them (i.e., the packet comes encapsulated in IPSEC, the kernel has IPSEC support, and can correctly decapsulate it).

Note that specifying **ipsec** is different from specifying **proto** *ipsec* as the latter will only look at the specific IP protocol field, irrespective of IPSEC kernel support and the validity of the IPSEC data.

Further note that this flag is silently ignored in kernels without IPSEC support. It does not affect rule processing when given and the rules are handled as if with no **ipsec** flag.

iptos *spec*
Matches IPv4 packets whose **tos** field contains the comma separated list of service types specified in *spec*. The supported IP types of service are:

lowdelay (IPTOS_LOWDELAY), **throughput** (IPTOS_THROUGHPUT), **reliability** (IPTOS_RELIABILITY), **mincost** (IPTOS_MINCOST), **congestion** (IPTOS_ECN_CE). The absence of a particular type may be denoted with a '!'.

dscp spec[,*spec*]

Matches IPv4/IPv6 packets whose **DS** field value is contained in *spec* mask. Multiple values can be specified via the comma separated list. Value can be one of keywords used in **setdscp** action or exact number.

ipttl *ttl-list*
Matches IPv4 packets whose time to live is included in *ttl-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

ipversion *ver*
Matches IP packets whose IP version field is *ver*.

keep-state [*:flowname*]
Upon a match, the firewall will create a dynamic rule, whose default behaviour is to match bidirectional traffic between source and destination IP/port using the same protocol. The rule has a limited lifetime (controlled by a set of sysctl(8) variables), and the lifetime is refreshed every time a matching packet is found. The *:flowname* is used to assign additional to addresses, ports and protocol parameter to dynamic rule. It can be used for more accurate matching by **check-state** rule. The **:default** keyword is special name used for compatibility with old rulesets.

layer2 Matches only layer2 packets, i.e., those passed to **ipfw** from **ether_demux**() and **ether_output_frame**().

limit {**src-addr** | **src-port** | **dst-addr** | **dst-port**} *N* [*:flowname*]
The firewall will only allow *N* connections with the same set of parameters as specified in the rule. One or more of source and destination addresses and ports can be specified.

lookup {**dst-ip** | **dst-port** | **dst-mac** | **src-ip** | **src-port** | **src-mac** | **uid** | **jail** | **dscp** | **mark**} *name*
Search an entry in lookup table *name* that matches the field specified as argument. If not found, the match fails. Otherwise, the match succeeds and **tablearg** is set to the value extracted from the table.

This option can be useful to quickly dispatch traffic based on certain packet fields. See the *LOOKUP TABLES* section below for more information on lookup tables.

{ MAC | mac } *dst-mac src-mac*
Match packets with a given *dst-mac* and *src-mac* addresses, specified as the **any** keyword (matching any MAC address), or six groups of hex digits separated by colons, and optionally followed by a mask indicating the significant bits. The mask may be specified using either of the following methods:

1. A slash (/) followed by the number of significant bits. For example, an address with 33 significant bits could be specified as:

   MAC 10:20:30:40:50:60/33 any

2. An ampersand (&) followed by a bitmask specified as six groups of hex digits separated by colons. For example, an address in which the last 16 bits are significant could be specified as:

   MAC 10:20:30:40:50:60&00:00:00:00:ff:ff any

   Note that the ampersand character has a special meaning in many shells and should generally be escaped.

Note that the order of MAC addresses (destination first, source second) is the same as on the wire, but the opposite of the one used for IP addresses.

mac-type *mac-type*

Matches packets whose Ethernet Type field corresponds to one of those specified as argument. *mac-type* is specified in the same way as **port numbers** (i.e., one or more comma-separated single values or ranges). You can use symbolic names for known values such as *vlan*, *ipv4, ipv6*. Values can be entered as decimal or hexadecimal (if prefixed by 0x), and they are always printed as hexadecimal (unless the **-N** option is used, in which case symbolic resolution will be attempted).

proto *protocol*

Matches packets with the corresponding IP protocol.

record-state

Upon a match, the firewall will create a dynamic rule as if **keep-state** was specified. However, this option doesn't imply an implicit **check-state** in contrast to **keep-state**.

recv | **xmit** | **via** {*ifX* | *ifmask* | *table*(*name*[,*value*]) | *ipno* | *any*}

Matches packets received, transmitted or going through, respectively, the interface specified by exact name (*ifX*), by device mask (*ifmask*), by IP address, or through some interface.

Interface name may be matched against *ifmask* with fnmatch(3) according to the rules used by the shell (f.e. tun*). See also the *EXAMPLES* section.

Table *name* may be used to match interface by its kernel ifindex. See the *LOOKUP TABLES* section below for more information on lookup tables.

The **via** keyword causes the interface to always be checked. If **recv** or **xmit** is used instead of **via**, then only the receive or transmit interface (respectively) is checked. By specifying both, it is possible to match packets based on both receive and transmit interface, e.g.:

```
ipfw add deny ip from any to any out recv ed0 xmit ed1
```

The **recv** interface can be tested on either incoming or outgoing packets, while the **xmit** interface can only be tested on outgoing packets. So **out** is required (and **in** is invalid) whenever **xmit** is used.

A packet might not have a receive or transmit interface: packets originating from the local host have no receive interface, while packets destined for the local host have no transmit interface.

set-limit {**src-addr** | **src-port** | **dst-addr** | **dst-port**} *N*
Works like **limit** but does not have an implicit **check-state** attached to it.

setup Matches TCP packets that have the SYN bit set but no ACK bit. This is the short form of "tcpflags syn,!ack".

sockarg
Matches packets that are associated to a local socket and for which the SO_USER_COOKIE socket option has been set to a non-zero value. As a side effect, the value of the option is made available as **tablearg** value, which in turn can be used as **skipto** or **pipe** number.

src-ip *ip-address*
Matches IPv4 packets whose source IP is one of the address(es) specified as an argument.

src-ip6 *ip6-address*
Matches IPv6 packets whose source IP is one of the address(es) specified as an argument.

src-port *ports*
Matches IP packets whose source port is one of the port(s) specified as argument.

tagged *tag-list*
Matches packets whose tags are included in *tag-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*. Tags can be applied to the packet using **tag** rule action parameter (see it's description for details on tags).

mark *value[:bitmask] | tablearg[:bitmask]*
Matches packets whose mark is equal to *value* with optional *bitmask* applied to it. **tablearg** can

also be used instead of an explicit *value* to match a value supplied by the last table lookup.

Both *value* and *bitmask* can be entered as decimal or hexadecimal (if prefixed by 0x), and they are always printed as hexadecimal.

tcpack *ack*
TCP packets only. Match if the TCP header acknowledgment number field is set to *ack*.

tcpdatalen *tcpdatalen-list*
Matches TCP packets whose length of TCP data is *tcpdatalen-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

tcpflags *spec*
TCP packets only. Match if the TCP header contains the comma separated list of flags specified in *spec*. The supported TCP flags are:

fin, **syn**, **rst**, **psh**, **ack** and **urg**. The absence of a particular flag may be denoted with a '!'. A rule which contains a **tcpflags** specification can never match a fragmented packet which has a non-zero offset. See the **frag** option for details on matching fragmented packets.

tcpmss *tcpmss-list*
Matches TCP packets whose MSS (maximum segment size) value is set to *tcpmss-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

tcpseq *seq*
TCP packets only. Match if the TCP header sequence number field is set to *seq*.

tcpwin *tcpwin-list*
Matches TCP packets whose header window field is set to *tcpwin-list*, which is either a single value or a list of values or ranges specified in the same way as *ports*.

tcpoptions *spec*
TCP packets only. Match if the TCP header contains the comma separated list of options specified in *spec*. The supported TCP options are:

mss (maximum segment size), **window** (tcp window advertisement), **sack** (selective ack), **ts** (rfc1323 timestamp) and **cc** (rfc1644 t/tcp connection count). The absence of a particular option may be denoted with a '!'.

uid *user*

Match all TCP or UDP packets sent by or received for a *user*. A *user* may be matched by name or identification number.

verrevpath

For incoming packets, a routing table lookup is done on the packet's source address. If the interface on which the packet entered the system matches the outgoing interface for the route, the packet matches. If the interfaces do not match up, the packet does not match. All outgoing packets or packets with no incoming interface match.

The name and functionality of the option is intentionally similar to the Cisco IOS command:

```
ip verify unicast reverse-path
```

This option can be used to make anti-spoofing rules to reject all packets with source addresses not from this interface. See also the option **antispoof**.

versrcreach

For incoming packets, a routing table lookup is done on the packet's source address. If a route to the source address exists, but not the default route or a blackhole/reject route, the packet matches. Otherwise, the packet does not match. All outgoing packets match.

The name and functionality of the option is intentionally similar to the Cisco IOS command:

```
ip verify unicast source reachable-via any
```

This option can be used to make anti-spoofing rules to reject all packets whose source address is unreachable.

antispoof

For incoming packets, the packet's source address is checked if it belongs to a directly connected network. If the network is directly connected, then the interface the packet came on in is compared to the interface the network is connected to. When incoming interface and directly connected interface are not the same, the packet does not match. Otherwise, the packet does match. All outgoing packets match.

This option can be used to make anti-spoofing rules to reject all packets that pretend to be from a directly connected network but do not come in through that interface. This option is similar to but more restricted than **verrevpath** because it engages only on packets with source addresses of directly connected networks instead of all source addresses.

LOOKUP TABLES

Lookup tables are useful to handle large sparse sets of addresses or other search keys (e.g., ports, jail IDs, interface names). In the rest of this section we will use the term ''key''. Table name needs to match the following spec: *table-name*. Tables with the same name can be created in different *sets*. However, rule links to the tables in *set 0* by default. This behavior can be controlled by *net.inet.ip.fw.tables_sets* variable. See the *SETS OF RULES* section for more information. There may be up to 65535 different lookup tables.

The following table types are supported:

table-type: *addr* | *iface* | *number* | *flow* | *mac*

table-key: *addr*[/*masklen*] | iface-name | number | flow-spec

flow-spec: *flow-field*[,*flow-spec*]

flow-field: *src-ip* | *proto* | *src-port* | *dst-ip* | *dst-port*

addr Matches IPv4 or IPv6 address. Each entry is represented by an *addr*[/*masklen*] and will match all addresses with base *addr* (specified as an IPv4/IPv6 address, or a hostname) and mask width of *masklen* bits. If *masklen* is not specified, it defaults to 32 for IPv4 and 128 for IPv6. When looking up an IP address in a table, the most specific entry will match.

iface Matches interface names. Each entry is represented by string treated as interface name. Wildcards are not supported.

number
Matches protocol ports, uids/gids or jail IDs. Each entry is represented by 32-bit unsigned integer. Ranges are not supported.

flow Matches packet fields specified by *flow* type suboptions with table entries.

mac Matches MAC address. Each entry is represented by an *addr*[/*masklen*] and will match all addresses with base *addr* and mask width of *masklen* bits. If *masklen* is not specified, it defaults to 48. When looking up an MAC address in a table, the most specific entry will match.

Tables require explicit creation via **create** before use.

The following creation options are supported:

create-options: *create-option* | *create-options*

create-option: **type** *table-type* | **valtype** *value-mask* | **algo** *algo-desc* |
limit *number* | **locked** | **missing** | **or-flush**

type Table key type.

valtype
Table value mask.

algo Table algorithm to use (see below).

limit Maximum number of items that may be inserted into table.

locked
Restrict any table modifications.

missing
Do not fail if table already exists and has exactly same options as new one.

or-flush
Flush existing table with same name instead of returning error. Implies **missing** so existing table must be compatible with new one.

Some of these options may be modified later via **modify** keyword. The following options can be changed:

modify-options: *modify-option* | *modify-options*

modify-option: **limit** *number*

limit Alter maximum number of items that may be inserted into table.

Additionally, table can be locked or unlocked using **lock** or **unlock** commands.

Tables of the same *type* can be swapped with each other using **swap** *name* command. Swap may fail if tables limits are set and data exchange would result in limits hit. Operation is performed atomically.

One or more entries can be added to a table at once using **add** command. Addition of all items are performed atomically. By default, error in addition of one entry does not influence addition of other

entries. However, non-zero error code is returned in that case. Special **atomic** keyword may be specified before **add** to indicate all-or-none add request.

One or more entries can be removed from a table at once using **delete** command. By default, error in removal of one entry does not influence removing of other entries. However, non-zero error code is returned in that case.

It may be possible to check what entry will be found on particular *table-key* using **lookup** *table-key* command. This functionality is optional and may be unsupported in some algorithms.

The following operations can be performed on *one* or **all** tables:

list List all entries.

flush Removes all entries.

info Shows generic table information.

detail Shows generic table information and algo-specific data.

The following lookup algorithms are supported:

algo-desc: *algo-name* | *algo-name algo-data*

algo-name: *addr: radix* | *addr: hash* | *iface: array* | *number: array* | *flow: hash* | *mac: radix*

addr: radix
Separate Radix trees for IPv4 and IPv6, the same way as the routing table (see route(4)). Default choice for *addr* type.

addr:hash
Separate auto-growing hashes for IPv4 and IPv6. Accepts entries with the same mask length specified initially via **addr:hash masks=/v4,/v6** algorithm creation options. Assume /32 and /128 masks by default. Search removes host bits (according to mask) from supplied address and checks resulting key in appropriate hash. Mostly optimized for /64 and byte-ranged IPv6 masks.

iface:array
Array storing sorted indexes for entries which are presented in the system. Optimized for very fast lookup.

number:array
Array storing sorted u32 numbers.

flow:hash
Auto-growing hash storing flow entries. Search calculates hash on required packet fields and searches for matching entries in selected bucket.

mac: radix
Radix tree for MAC address

The **tablearg** feature provides the ability to use a value, looked up in the table, as the argument for a rule action, action parameter or rule option. This can significantly reduce number of rules in some configurations. If two tables are used in a rule, the result of the second (destination) is used.

Each record may hold one or more values according to *value-mask*. This mask is set on table creation via **valtype** option. The following value types are supported:

value-mask: *value-type*[,*value-mask*]

value-type: *skipto* | *pipe* | *fib* | *nat* | *dscp* | *tag* | *divert* |
netgraph | *limit* | *ipv4* | *ipv6* | *mark*

skipto rule number to jump to.

pipe Pipe number to use.

fib fib number to match/set.

nat nat number to jump to.

dscp dscp value to match/set.

tag tag number to match/set.

divert port number to divert traffic to.

netgraph
hook number to move packet to.

limit maximum number of connections.

ipv4 IPv4 nexthop to fwd packets to.

ipv6 IPv6 nexthop to fwd packets to.

mark mark value to match/set.

The **tablearg** argument can be used with the following actions: **nat, pipe, queue, divert, tee, netgraph, ngtee, fwd, skipto, setfib**, **setmark**, action parameters: **tag, untag**, rule options: **limit, tagged, mark**.

When used with the **skipto** action, the user should be aware that the code will walk the ruleset up to a rule equal to, or past, the given number.

See the *EXAMPLES* Section for example usage of tables and the tablearg keyword.

SETS OF RULES

Each rule or table belongs to one of 32 different *sets* , numbered 0 to 31. Set 31 is reserved for the default rule.

By default, rules or tables are put in set 0, unless you use the **set N** attribute when adding a new rule or table. Sets can be individually and atomically enabled or disabled, so this mechanism permits an easy way to store multiple configurations of the firewall and quickly (and atomically) switch between them.

By default, tables from set 0 are referenced when adding rule with table opcodes regardless of rule set. This behavior can be changed by setting *net.inet.ip.fw.tables_sets* variable to 1. Rule's set will then be used for table references.

The command to enable/disable sets is

> **ipfw set** [**disable** *number* ...] [**enable** *number* ...]

where multiple **enable** or **disable** sections can be specified. Command execution is atomic on all the sets specified in the command. By default, all sets are enabled.

When you disable a set, its rules behave as if they do not exist in the firewall configuration, with only one exception:

> dynamic rules created from a rule before it had been disabled will still be active until they expire. In order to delete dynamic rules you have to explicitly delete the parent rule which generated them.

The set number of rules can be changed with the command

ipfw set move {**rule** *rule-number* | *old-set*} **to** *new-set*

Also, you can atomically swap two rulesets with the command

ipfw set swap *first-set second-set*

See the *EXAMPLES* Section on some possible uses of sets of rules.

STATEFUL FIREWALL

Stateful operation is a way for the firewall to dynamically create rules for specific flows when packets that match a given pattern are detected. Support for stateful operation comes through the **check-state**, **keep-state**, **record-state**, **limit** and **set-limit** options of **rules**.

Dynamic rules are created when a packet matches a **keep-state**, **record-state**, **limit** or **set-limit** rule, causing the creation of a *dynamic* rule which will match all and only packets with a given *protocol* between a *src-ip/src-port dst-ip/dst-port* pair of addresses (*src* and *dst* are used here only to denote the initial match addresses, but they are completely equivalent afterwards). Rules created by **keep-state** option also have a *:flowname* taken from it. This name is used in matching together with addresses, ports and protocol. Dynamic rules will be checked at the first **check-state, keep-state** or **limit** occurrence, and the action performed upon a match will be the same as in the parent rule.

Note that no additional attributes other than protocol and IP addresses and ports and :flowname are checked on dynamic rules.

The typical use of dynamic rules is to keep a closed firewall configuration, but let the first TCP SYN packet from the inside network install a dynamic rule for the flow so that packets belonging to that session will be allowed through the firewall:

```
ipfw add check-state :OUTBOUND
ipfw add allow tcp from my-subnet to any setup keep-state :OUTBOUND
ipfw add deny tcp from any to any
```

A similar approach can be used for UDP, where an UDP packet coming from the inside will install a dynamic rule to let the response through the firewall:

```
ipfw add check-state :OUTBOUND
ipfw add allow udp from my-subnet to any keep-state :OUTBOUND
ipfw add deny udp from any to any
```

Dynamic rules expire after some time, which depends on the status of the flow and the setting of some

sysctl variables. See Section *SYSCTL VARIABLES* for more details. For TCP sessions, dynamic rules can be instructed to periodically send keepalive packets to refresh the state of the rule when it is about to expire.

See Section *EXAMPLES* for more examples on how to use dynamic rules.

TRAFFIC SHAPER (DUMMYNET) CONFIGURATION

ipfw is also the user interface for the **dummynet** traffic shaper, packet scheduler and network emulator, a subsystem that can artificially queue, delay or drop packets emulating the behaviour of certain network links or queueing systems.

dummynet operates by first using the firewall to select packets using any match pattern that can be used in **ipfw** rules. Matching packets are then passed to either of two different objects, which implement the traffic regulation:

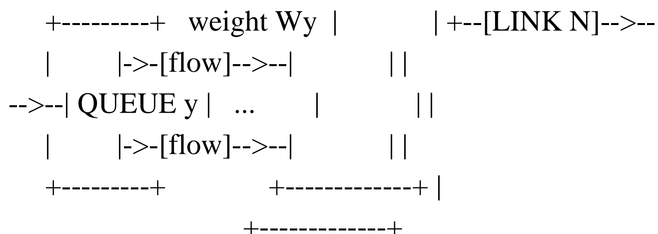
pipe A *pipe* emulates a *link* with given bandwidth and propagation delay, driven by a FIFO scheduler and a single queue with programmable queue size and packet loss rate. Packets are appended to the queue as they come out from **ipfw**, and then transferred in FIFO order to the link at the desired rate.

queue A *queue* is an abstraction used to implement packet scheduling using one of several packet scheduling algorithms. Packets sent to a *queue* are first grouped into flows according to a mask on the 5-tuple. Flows are then passed to the scheduler associated to the *queue*, and each flow uses scheduling parameters (weight and others) as configured in the *queue* itself. A scheduler in turn is connected to an emulated link, and arbitrates the link's bandwidth among backlogged flows according to weights and to the features of the scheduling algorithm in use.

In practice, *pipes* can be used to set hard limits to the bandwidth that a flow can use, whereas *queues* can be used to determine how different flows share the available bandwidth.

A graphical representation of the binding of queues, flows, schedulers and links is below.

```
             (flow_mask|sched_mask)  sched_mask
     +---------+   weight Wx  +-------------+
     |         |->-[flow]-->--|             |-+
  -->--| QUEUE x |   ...        |             | |
     |         |->-[flow]-->--| SCHEDuler N | |
     +---------+              |             | |
         ...                  |             +--[LINK N]-->--
```



It is important to understand the role of the SCHED_MASK and FLOW_MASK, which are configured through the commands

```
ipfw sched N config mask SCHED_MASK ...
```

and

```
ipfw queue X config mask FLOW_MASK ....
```

The SCHED_MASK is used to assign flows to one or more scheduler instances, one for each value of the packet's 5-tuple after applying SCHED_MASK. As an example, using ‘‘src-ip 0xffffff00’’ creates one instance for each /24 destination subnet.

The FLOW_MASK, together with the SCHED_MASK, is used to split packets into flows. As an example, using ‘‘src-ip 0x000000ff’’ together with the previous SCHED_MASK makes a flow for each individual source address. In turn, flows for each /24 subnet will be sent to the same scheduler instance.

The above diagram holds even for the *pipe* case, with the only restriction that a *pipe* only supports a SCHED_MASK, and forces the use of a FIFO scheduler (these are for backward compatibility reasons; in fact, internally, a **dummynet's** pipe is implemented exactly as above).

There are two modes of **dummynet** operation: "normal" and "fast". The "normal" mode tries to emulate a real link: the **dummynet** scheduler ensures that the packet will not leave the pipe faster than it would on the real link with a given bandwidth. The "fast" mode allows certain packets to bypass the **dummynet** scheduler (if packet flow does not exceed pipe's bandwidth). This is the reason why the "fast" mode requires less CPU cycles per packet (on average) and packet latency can be significantly lower in comparison to a real link with the same bandwidth. The default mode is "normal". The "fast" mode can be enabled by setting the *net.inet.ip.dummynet.io_fast* sysctl(8) variable to a non-zero value.

PIPE, QUEUE AND SCHEDULER CONFIGURATION

The *pipe*, *queue* and *scheduler* configuration commands are the following:

pipe *number* **config** *pipe-configuration*

queue *number* **config** *queue-configuration*

sched *number* **config** *sched-configuration*

The following parameters can be configured for a pipe:

bw *bandwidth* | *device*

Bandwidth, measured in [**K**|**M**|**G**]{**bit/s**|**Byte/s**}.

A value of 0 (default) means unlimited bandwidth. The unit must immediately follow the number, as in

```
dnctl pipe 1 config bw 300Kbit/s
```

If a device name is specified instead of a numeric value, as in

```
dnctl pipe 1 config bw tun0
```

then the transmit clock is supplied by the specified device. At the moment only the tun(4) device supports this functionality, for use in conjunction with ppp(8).

delay *ms-delay*

Propagation delay, measured in milliseconds. The value is rounded to the next multiple of the clock tick (typically 10ms, but it is a good practice to run kernels with "options HZ=1000" to reduce the granularity to 1ms or less). The default value is 0, meaning no delay.

burst *size*

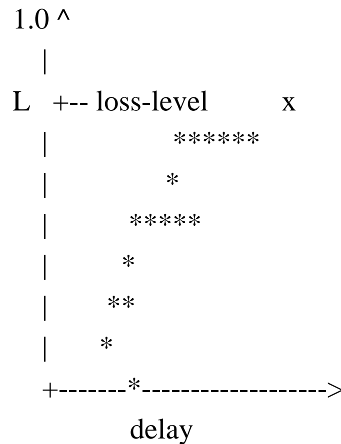
If the data to be sent exceeds the pipe's bandwidth limit (and the pipe was previously idle), up to *size* bytes of data are allowed to bypass the **dummynet** scheduler, and will be sent as fast as the physical link allows. Any additional data will be transmitted at the rate specified by the **pipe** bandwidth. The burst size depends on how long the pipe has been idle; the effective burst size is calculated as follows: MAX(*size* , **bw** * pipe_idle_time).

profile *filename*

A file specifying the additional overhead incurred in the transmission of a packet on the link.

Some link types introduce extra delays in the transmission of a packet, e.g., because of MAC level framing, contention on the use of the channel, MAC level retransmissions and so on. From our point of view, the channel is effectively unavailable for this extra time, which is constant or variable depending on the link type. Additionally, packets may be dropped after this time (e.g., on a wireless link after too many retransmissions). We can model the additional delay with an empirical curve that represents its distribution.

cumulative probability



The empirical curve may have both vertical and horizontal lines. Vertical lines represent constant delay for a range of probabilities. Horizontal lines correspond to a discontinuity in the delay distribution: the pipe will use the largest delay for a given probability.

The file format is the following, with whitespace acting as a separator and '#' indicating the beginning a comment:

name *identifier*
: optional name (listed by "dnctl pipe show") to identify the delay distribution;

bw *value*
: the bandwidth used for the pipe. If not specified here, it must be present explicitly as a configuration parameter for the pipe;

loss-level *L*
: the probability above which packets are lost. (0.0 <= L <= 1.0, default 1.0 i.e., no loss);

samples *N*
: the number of samples used in the internal representation of the curve (2..1024; default 100);

delay prob | prob delay
: One of these two lines is mandatory and defines the format of the following lines with data points.

XXX YYY
: 2 or more lines representing points in the curve, with either delay or probability first, according to the chosen format. The unit for delay is milliseconds. Data points do not need to be sorted. Also, the number of actual lines can be different from the value of the

"samples" parameter: **ipfw** utility will sort and interpolate the curve as needed.

Example of a profile file:

```
name    bla_bla_bla
samples 100
loss-level    0.86
prob    delay
0       200 # minimum overhead is 200ms
0.5     200
0.5     300
0.8     1000
0.9     1300
1       1300
#configuration file end
```

The following parameters can be configured for a queue:

pipe *pipe_nr*
Connects a queue to the specified pipe. Multiple queues (with the same or different weights) can be connected to the same pipe, which specifies the aggregate rate for the set of queues.

weight *weight*
Specifies the weight to be used for flows matching this queue. The weight must be in the range 1..100, and defaults to 1.

The following case-insensitive parameters can be configured for a scheduler:

type *{fifo | wf2q+ | rr | qfq | fq_codel | fq_pie}*
specifies the scheduling algorithm to use.

fifo is just a FIFO scheduler (which means that all packets are stored in the same queue as they arrive to the scheduler). FIFO has O(1) per-packet time complexity, with very low constants (estimate 60-80ns on a 2GHz desktop machine) but gives no service guarantees.

wf2q+
implements the WF2Q+ algorithm, which is a Weighted Fair Queueing algorithm which permits flows to share bandwidth according to their weights. Note that weights are not priorities; even a flow with a minuscule weight will never starve. WF2Q+ has O(log N) per-packet processing cost, where N is the number of flows, and is the default algorithm used by previous versions dummynet's queues.

rr implements the Deficit Round Robin algorithm, which has O(1) processing costs

(roughly, 100-150ns per packet) and permits bandwidth allocation according to weights, but with poor service guarantees.

qfq implements the QFQ algorithm, which is a very fast variant of WF2Q+, with similar service guarantees and O(1) processing costs (roughly, 200-250ns per packet).

fq_codel
implements the FQ-CoDel (FlowQueue-CoDel) scheduler/AQM algorithm, which uses a modified Deficit Round Robin scheduler to manage two lists of sub-queues (old sub-queues and new sub-queues) for providing brief periods of priority to lightweight or short burst flows. By default, the total number of sub-queues is 1024. FQ-CoDel's internal, dynamically created sub-queues are controlled by separate instances of CoDel AQM.

fq_pie
implements the FQ-PIE (FlowQueue-PIE) scheduler/AQM algorithm, which similar to **fq_codel** but uses per sub-queue PIE AQM instance to control the queue delay.

fq_codel inherits AQM parameters and options from **codel** (see below), and **fq_pie** inherits AQM parameters and options from **pie** (see below). Additionally, both of **fq_codel** and **fq_pie** have shared scheduler parameters which are:

quantum
m specifies the quantum (credit) of the scheduler. *m* is the number of bytes a queue can serve before being moved to the tail of old queues list. The default is 1514 bytes, and the maximum acceptable value is 9000 bytes.

limit *m* specifies the hard size limit (in unit of packets) of all queues managed by an instance of the scheduler. The default value of *m* is 10240 packets, and the maximum acceptable value is 20480 packets.

flows *m* specifies the total number of flow queues (sub-queues) that fq_* creates and manages. By default, 1024 sub-queues are created when an instance of the fq_{codel/pie} scheduler is created. The maximum acceptable value is 65536.

Note that any token after **fq_codel** or **fq_pie** is considered a parameter for fq_{codel/pie}. So, ensure all scheduler configuration options not related to fq_{codel/pie} are written before **fq_codel/fq_pie** tokens.

In addition to the type, all parameters allowed for a pipe can also be specified for a scheduler.

Finally, the following parameters can be configured for both pipes and queues:

buckets *hash-table-size*

Specifies the size of the hash table used for storing the various queues. Default value is 64 controlled by the sysctl(8) variable *net.inet.ip.dummynet.hash_size*, allowed range is 16 to 65536.

mask *mask-specifier*

Packets sent to a given pipe or queue by an **ipfw** rule can be further classified into multiple flows, each of which is then sent to a different *dynamic* pipe or queue. A flow identifier is constructed by masking the IP addresses, ports and protocol types as specified with the **mask** options in the configuration of the pipe or queue. For each different flow identifier, a new pipe or queue is created with the same parameters as the original object, and matching packets are sent to it.

Thus, when *dynamic pipes* are used, each flow will get the same bandwidth as defined by the pipe, whereas when *dynamic queues* are used, each flow will share the parent's pipe bandwidth evenly with other flows generated by the same queue (note that other queues with different weights might be connected to the same pipe).
Available mask specifiers are a combination of one or more of the following:

dst-ip *mask*, **dst-ip6** *mask*, **src-ip** *mask*, **src-ip6** *mask*, **dst-port** *mask*, **src-port** *mask*, **flow-id** *mask*, **proto** *mask* or **all**,

where the latter means all bits in all fields are significant.

noerror

When a packet is dropped by a **dummynet** queue or pipe, the error is normally reported to the caller routine in the kernel, in the same way as it happens when a device queue fills up. Setting this option reports the packet as successfully delivered, which can be needed for some experimental setups where you want to simulate loss or congestion at a remote router.

plr *packet-loss-rate*

Packet loss rate. Argument *packet-loss-rate* is a floating-point number between 0 and 1, with 0 meaning no loss, 1 meaning 100% loss. The loss rate is internally represented on 31 bits.

queue {*slots* | *size***Kbytes**}

Queue size, in *slots* or **KBytes**. Default value is 50 slots, which is the typical queue size for Ethernet devices. Note that for slow speed links you should keep the queue size short or your traffic might be affected by a significant queueing delay. E.g., 50 max-sized Ethernet packets (1500 bytes) mean 600Kbit or 20s of queue on a 30Kbit/s pipe. Even worse effects can result if you get packets from an interface with a much larger MTU, e.g. the loopback interface with its 16KB packets. The sysctl(8) variables *net.inet.ip.dummynet.pipe_byte_limit* and

net.inet.ip.dummynet.pipe_slot_limit control the maximum lengths that can be specified.

red | **gred** *w_q*/*min_th*/*max_th*/*max_p*
[ecn] Make use of the RED (Random Early Detection) queue management algorithm. *w_q* and *max_p* are floating point numbers between 0 and 1 (inclusive), while *min_th* and *max_th* are integer numbers specifying thresholds for queue management (thresholds are computed in bytes if the queue has been defined in bytes, in slots otherwise). The two parameters can also be of the same value if needed. The **dummynet** also supports the gentle RED variant (gred) and ECN (Explicit Congestion Notification) as optional. Three sysctl(8) variables can be used to control the RED behaviour:

net.inet.ip.dummynet.red_lookup_depth
specifies the accuracy in computing the average queue when the link is idle (defaults to 256, must be greater than zero)

net.inet.ip.dummynet.red_avg_pkt_size
specifies the expected average packet size (defaults to 512, must be greater than zero)

net.inet.ip.dummynet.red_max_pkt_size
specifies the expected maximum packet size, only used when queue thresholds are in bytes (defaults to 1500, must be greater than zero).

codel [**target** *time*] [**interval** *time*] [**ecn** | **noecn**]
Make use of the CoDel (Controlled-Delay) queue management algorithm. *time* is interpreted as milliseconds by default but seconds (s), milliseconds (ms) or microseconds (us) can be specified instead. CoDel drops or marks (ECN) packets depending on packet sojourn time in the queue. **target** *time* (5ms by default) is the minimum acceptable persistent queue delay that CoDel allows. CoDel does not drop packets directly after packets sojourn time becomes higher than **target** *time* but waits for **interval** *time* (100ms default) before dropping. **interval** *time* should be set to maximum RTT for all expected connections. **ecn** enables (disabled by default) packet marking (instead of dropping) for ECN-enabled TCP flows when queue delay becomes high.

Note that any token after **codel** is considered a parameter for CoDel. So, ensure all pipe/queue configuration options are written before **codel** token.

The sysctl(8) variables *net.inet.ip.dummynet.codel.target* and *net.inet.ip.dummynet.codel.interval* can be used to set CoDel default parameters.

pie [**target** *time*] [**tupdate** *time*] [**alpha** *n*] [**beta** *n*] [**max_burst** *time*] [**max_ecnth** *n*] [**ecn** | **noecn**] [**capdrop** | **nocapdrop**] [**drand** | **nodrand**] [**onoff**] [**dre** | **ts**]

Make use of the PIE (Proportional Integral controller Enhanced) queue management algorithm. PIE drops or marks packets depending on a calculated drop probability during en-queue process, with the aim of achieving high throughput while keeping queue delay low. At regular time intervals of **tupdate** *time* (15ms by default) a background process (re)calculates the probability based on queue delay deviations from **target** *time* (15ms by default) and queue delay trends. PIE approximates current queue delay by using a departure rate estimation method, or (optionally) by using a packet timestamp method similar to CoDel. *time* is interpreted as milliseconds by default but seconds (s), milliseconds (ms) or microseconds (us) can be specified instead. The other PIE parameters and options are as follows:

alpha *n*
: *n* is a floating point number between 0 and 7 which specifies the weight of queue delay deviations that is used in drop probability calculation. 0.125 is the default.

beta *n*
: *n* is a floating point number between 0 and 7 which specifies is the weight of queue delay trend that is used in drop probability calculation. 1.25 is the default.

max_burst *time*
: The maximum period of time that PIE does not drop/mark packets. 150ms is the default and 10s is the maximum value.

max_ecnth *n*
: Even when ECN is enabled, PIE drops packets instead of marking them when drop probability becomes higher than ECN probability threshold **max_ecnth** *n* , the default is 0.1 (i.e 10%) and 1 is the maximum value.

ecn | **noecn**
: enable or disable ECN marking for ECN-enabled TCP flows. Disabled by default.

capdrop | **nocapdrop**
: enable or disable cap drop adjustment. Cap drop adjustment is enabled by default.

drand | **nodrand**
: enable or disable drop probability de-randomisation. De-randomisation eliminates the problem of dropping packets too close or too far. De-randomisation is enabled by default.

onoff
: enable turning PIE on and off depending on queue load. If this option is enabled, PIE turns on when over 1/3 of queue becomes full. This option is disabled by default.

dre | **ts**

Calculate queue delay using departure rate estimation **dre** or timestamps **ts**. **dre** is used by default.

Note that any token after **pie** is considered a parameter for PIE. So ensure all pipe/queue the configuration options are written before **pie** token. sysctl(8) variables can be used to control the **pie** default parameters. See the *SYSCTL VARIABLES* section for more details.

When used with IPv6 data, **dummynet** currently has several limitations. Information necessary to route link-local packets to an interface is not available after processing by **dummynet** so those packets are dropped in the output path. Care should be taken to ensure that link-local packets are not passed to **dummynet**.

CHECKLIST

Here are some important points to consider when designing your rules:

- Remember that you filter both packets going **in** and **out**. Most connections need packets going in both directions.

- Remember to test very carefully. It is a good idea to be near the console when doing this. If you cannot be near the console, use an auto-recovery script such as the one in */usr/share/examples/ipfw/change_rules.sh*.

- Do not forget the loopback interface.

FINE POINTS

- There are circumstances where fragmented datagrams are unconditionally dropped. TCP packets are dropped if they do not contain at least 20 bytes of TCP header, UDP packets are dropped if they do not contain a full 8 byte UDP header, and ICMP packets are dropped if they do not contain 4 bytes of ICMP header, enough to specify the ICMP type, code, and checksum. These packets are simply logged as "pullup failed" since there may not be enough good data in the packet to produce a meaningful log entry.

- Another type of packet is unconditionally dropped, a TCP packet with a fragment offset of one. This is a valid packet, but it only has one use, to try to circumvent firewalls. When logging is enabled, these packets are reported as being dropped by rule -1.

- If you are logged in over a network, loading the kld(4) version of **ipfw** is probably not as straightforward as you would think. The following command line is recommended:

```
kldload ipfw && \
ipfw add 32000 allow ip from any to any
```

Along the same lines, doing an

```
ipfw flush
```

in similar surroundings is also a bad idea.

- The **ipfw** filter list may not be modified if the system security level is set to 3 or higher (see init(8) for information on system security levels).

PACKET DIVERSION

A divert(4) socket bound to the specified port will receive all packets diverted to that port. If no socket is bound to the destination port, or if the divert module is not loaded, or if the kernel was not compiled with divert socket support, the packets are dropped.

NETWORK ADDRESS TRANSLATION (NAT)

ipfw support in-kernel NAT using the kernel version of libalias(3). The kernel module **ipfw_nat** should be loaded or kernel should have **options IPFIREWALL_NAT** to be able use NAT.

The nat configuration command is the following:

nat *nat_number* **config** *nat-configuration*

The following parameters can be configured:

ip *ip_address*
Define an ip address to use for aliasing.

if *nic* Use ip address of NIC for aliasing, dynamically changing it if NIC's ip address changes.

log Enable logging on this nat instance.

deny_in
Deny any incoming connection from outside world.

same_ports
Try to leave the alias port numbers unchanged from the actual local port numbers.

unreg_only
Traffic on the local network not originating from a RFC 1918 unregistered address spaces will be ignored.

unreg_cgn
Like unreg_only, but includes the RFC 6598 (Carrier Grade NAT) address range.

reset Reset table of the packet aliasing engine on address change.

reverse
Reverse the way libalias handles aliasing.

proxy_only
Obey transparent proxy rules only, packet aliasing is not performed.

skip_global
Skip instance in case of global state lookup (see below).

port_range *lower-upper*
Set the aliasing ports between the ranges given. Upper port has to be greater than lower.

Some special values can be supplied instead of *nat_number* in nat rule actions:

global Looks up translation state in all configured nat instances. If an entry is found, packet is aliased according to that entry. If no entry was found in any of the instances, packet is passed unchanged, and no new entry will be created. See section *MULTIPLE INSTANCES* in natd(8) for more information.

tablearg
Uses argument supplied in lookup table. See *LOOKUP TABLES* section below for more information on lookup tables.

To let the packet continue after being (de)aliased, set the sysctl variable *net.inet.ip.fw.one_pass* to 0. For more information about aliasing modes, refer to libalias(3). See Section *EXAMPLES* for some examples of nat usage.

REDIRECT AND LSNAT SUPPORT IN IPFW

Redirect and LSNAT support follow closely the syntax used in natd(8). See Section *EXAMPLES* for some examples on how to do redirect and lsnat.

SCTP NAT SUPPORT

SCTP nat can be configured in a similar manner to TCP through the **ipfw** command line tool. The main difference is that **sctp nat** does not do port translation. Since the local and global side ports will be the same, there is no need to specify both. Ports are redirected as follows:

nat *nat_number* **config if** *nic* **redirect_port sctp** *ip_address [,addr_list] {[port | port-port] [,ports]}*

Most **sctp nat** configuration can be done in real-time through the sysctl(8) interface. All may be changed dynamically, though the hash_table size will only change for new **nat** instances. See *SYSCTL VARIABLES* for more info.

IPv6/IPv4 NETWORK ADDRESS AND PROTOCOL TRANSLATION

Stateful translation

ipfw supports in-kernel IPv6/IPv4 network address and protocol translation. Stateful NAT64 translation allows IPv6-only clients to contact IPv4 servers using unicast TCP, UDP or ICMP protocols. One or more IPv4 addresses assigned to a stateful NAT64 translator are shared among several IPv6-only clients. When stateful NAT64 is used in conjunction with DNS64, no changes are usually required in the IPv6 client or the IPv4 server. The kernel module **ipfw_nat64** should be loaded or kernel should have **options IPFIREWALL_NAT64** to be able use stateful NAT64 translator.

Stateful NAT64 uses a bunch of memory for several types of objects. When IPv6 client initiates connection, NAT64 translator creates a host entry in the states table. Each host entry uses preallocated IPv4 alias entry. Each alias entry has a number of ports group entries allocated on demand. Ports group entries contains connection state entries. There are several options to control limits and lifetime for these objects.

NAT64 translator follows RFC7915 when does ICMPv6/ICMP translation, unsupported message types will be silently dropped. IPv6 needs several ICMPv6 message types to be explicitly allowed for correct operation. Make sure that ND6 neighbor solicitation (ICMPv6 type 135) and neighbor advertisement (ICMPv6 type 136) messages will not be handled by translation rules.

After translation NAT64 translator by default sends packets through corresponding netisr queue. Thus translator host should be configured as IPv4 and IPv6 router. Also this means, that a packet is handled by firewall twice. First time an original packet is handled and consumed by translator, and then it is handled again as translated packet. This behavior can be changed by sysctl variable *net.inet.ip.fw.nat64_direct_output*. Also translated packet can be tagged using **tag** rule action, and then matched by **tagged** opcode to avoid loops and extra overhead.

The stateful NAT64 configuration command is the following:

nat64lsn *name* **create** *create-options*

The following parameters can be configured:

prefix4 *ipv4_prefix/plen*
The IPv4 prefix with mask defines the pool of IPv4 addresses used as source address after translation. Stateful NAT64 module translates IPv6 source address of client to one IPv4 address from this pool. Note that incoming IPv4 packets that don't have corresponding state entry in the states table will be dropped by translator. Make sure that translation rules handle packets, destined to configured prefix.

prefix6 *ipv6_prefix/length*
The IPv6 prefix defines IPv4-embedded IPv6 addresses used by translator to represent IPv4 addresses. This IPv6 prefix should be configured in DNS64. The translator implementation follows RFC6052, that restricts the length of prefixes to one of following: 32, 40, 48, 56, 64, or 96. The Well-Known IPv6 Prefix 64:ff9b:: must be 96 bits long. The special *::/length* prefix can be used to handle several IPv6 prefixes with one NAT64 instance. The NAT64 instance will determine a destination IPv4 address from prefix *length*.

states_chunks *number*
The number of states chunks in single ports group. Each ports group by default can keep 64 state entries in single chunk. The above value affects the maximum number of states that can be associated with single IPv4 alias address and port. The value must be power of 2, and up to 128.

host_del_age *seconds*
The number of seconds until the host entry for a IPv6 client will be deleted and all its resources will be released due to inactivity. Default value is *3600*.

pg_del_age *seconds*
The number of seconds until a ports group with unused state entries will be released. Default value is *900*.

tcp_syn_age *seconds*
The number of seconds while a state entry for TCP connection with only SYN sent will be kept. If TCP connection establishing will not be finished, state entry will be deleted. Default value is *10*.

tcp_est_age *seconds*
The number of seconds while a state entry for established TCP connection will be kept. Default value is *7200*.

tcp_close_age *seconds*
The number of seconds while a state entry for closed TCP connection will be kept. Keeping state entries for closed connections is needed, because IPv4 servers typically keep closed connections in a TIME_WAIT state for a several minutes. Since translator's IPv4 addresses are shared among all IPv6 clients, new connections from the same addresses and ports may be rejected by server, because these connections are still in a TIME_WAIT state. Keeping them in translator's state table protects from such rejects. Default value is *180*.

udp_age *seconds*
The number of seconds while translator keeps state entry in a waiting for reply to the sent UDP datagram. Default value is *120*.

icmp_age *seconds*
The number of seconds while translator keeps state entry in a waiting for reply to the sent ICMP message. Default value is *60*.

log Turn on logging of all handled packets via BPF through *ipfwlog0* interface. *ipfwlog0* is a pseudo interface and can be created after a boot manually with **ifconfig** command. Note that it has different purpose than *ipfw0* interface. Translators sends to BPF an additional information with each packet. With **tcpdump** you are able to see each handled packet before and after translation.

-log Turn off logging of all handled packets via BPF.

allow_private
Turn on processing private IPv4 addresses. By default IPv6 packets with destinations mapped to private address ranges defined by RFC1918 are not processed.

-allow_private
Turn off private address handling in **nat64** instance.

To inspect a states table of stateful NAT64 the following command can be used:

nat64lsn *name* **show states**

Stateless NAT64 translator doesn't use a states table for translation and converts IPv4 addresses to IPv6 and vice versa solely based on the mappings taken from configured lookup tables. Since a states table doesn't used by stateless translator, it can be configured to pass IPv4 clients to IPv6-only servers.

The stateless NAT64 configuration command is the following:

nat64stl *name* **create** *create-options*

The following parameters can be configured:

prefix6 *ipv6_prefix/length*
The IPv6 prefix defines IPv4-embedded IPv6 addresses used by translator to represent IPv4 addresses. This IPv6 prefix should be configured in DNS64.

table4 *table46*
The lookup table *table46* contains mapping how IPv4 addresses should be translated to IPv6 addresses.

table6 *table64*
The lookup table *table64* contains mapping how IPv6 addresses should be translated to IPv4 addresses.

log Turn on logging of all handled packets via BPF through *ipfwlog0* interface.

-log Turn off logging of all handled packets via BPF.

allow_private
Turn on processing private IPv4 addresses. By default IPv6 packets with destinations mapped to private address ranges defined by RFC1918 are not processed.

-allow_private
Turn off private address handling in **nat64** instance.

Note that the behavior of stateless translator with respect to not matched packets differs from stateful translator. If corresponding addresses was not found in the lookup tables, the packet will not be dropped and the search continues.

XLAT464 CLAT translation

XLAT464 CLAT NAT64 translator implements client-side stateless translation as defined in RFC6877 and is very similar to statless NAT64 translator explained above. Instead of lookup tables it uses one-to-one mapping between IPv4 and IPv6 addresses using configured prefixes. This mode can be used as a replacement of DNS64 service for applications that are not using it (e.g. VoIP) allowing them to access IPv4-only Internet over IPv6-only networks with help of remote NAT64 translator.

The CLAT NAT64 configuration command is the following:

nat64clat *name* **create** *create-options*

The following parameters can be configured:

clat_prefix *ipv6_prefix/length*
The IPv6 prefix defines IPv4-embedded IPv6 addresses used by translator to represent source IPv4 addresses.

plat_prefix *ipv6_prefix/length*
The IPv6 prefix defines IPv4-embedded IPv6 addresses used by translator to represent destination IPv4 addresses. This IPv6 prefix should be configured on a remote NAT64 translator.

log Turn on logging of all handled packets via BPF through *ipfwlog0* interface.

-log Turn off logging of all handled packets via BPF.

allow_private
Turn on processing private IPv4 addresses. By default **nat64clat** instance will not process IPv4 packets with destination address from private ranges as defined in RFC1918.

-allow_private
Turn off private address handling in **nat64clat** instance.

Note that the behavior of CLAT translator with respect to not matched packets differs from stateful translator. If corresponding addresses were not matched against prefixes configured, the packet will not be dropped and the search continues.

IPv6-to-IPv6 NETWORK PREFIX TRANSLATION (NPTv6)

ipfw supports in-kernel IPv6-to-IPv6 network prefix translation as described in RFC6296. The kernel module **ipfw_nptv6** should be loaded or kernel should has **options IPFIREWALL_NPTV6** to be able use NPTv6 translator.

The NPTv6 configuration command is the following:

nptv6 *name* **create** *create-options*

The following parameters can be configured:

int_prefix *ipv6_prefix*

IPv6 prefix used in internal network. NPTv6 module translates source address when it matches this prefix.

ext_prefix *ipv6_prefix*
IPv6 prefix used in external network. NPTv6 module translates destination address when it matches this prefix.

ext_if *nic*
The NPTv6 module will use first global IPv6 address from interface *nic* as external prefix. It can be useful when IPv6 prefix of external network is dynamically obtained. **ext_prefix** and **ext_if** options are mutually exclusive.

prefixlen *length*
The length of specified IPv6 prefixes. It must be in range from 8 to 64.

Note that the prefix translation rules are silently ignored when IPv6 packet forwarding is disabled. To enable the packet forwarding, set the sysctl variable *net.inet6.ip6.forwarding* to 1.

To let the packet continue after being translated, set the sysctl variable *net.inet.ip.fw.one_pass* to 0.

LOADER TUNABLES

Tunables can be set in loader(8) prompt, loader.conf(5) or kenv(1) before ipfw module gets loaded.

net.inet.ip.fw.enable: 1
Enables the firewall. Setting this variable to 0 lets you run your machine without firewall even if compiled in.

net.inet6.ip6.fw.enable: 1
provides the same functionality as above for the IPv6 case.

net.link.ether.ipfw: 0
Controls whether layer2 packets are passed to **ipfw**. Default is no.

net.inet.ip.fw.default_to_accept: 0
Defines ipfw last rule behavior. This value overrides **options IPFW_DEFAULT_TO_(ACCEPT|DENY)** from kernel configuration file.

net.inet.ip.fw.tables_max: 128
Defines number of tables available in ipfw. Number cannot exceed 65534.

SYSCTL VARIABLES

A set of sysctl(8) variables controls the behaviour of the firewall and associated modules (**dummynet**, **bridge**, **sctp nat**). These are shown below together with their default value (but always check with the sysctl(8) command what value is actually in use) and meaning:

net.inet.ip.alias.sctp.accept_global_ootb_addip: 0
Defines how the **nat** responds to receipt of global OOTB ASCONF-AddIP:

0 No response (unless a partially matching association exists - ports and vtags match but global address does not)

1 **nat** will accept and process all OOTB global AddIP messages.

Option 1 should never be selected as this forms a security risk. An attacker can establish multiple fake associations by sending AddIP messages.

net.inet.ip.alias.sctp.chunk_proc_limit: 5
Defines the maximum number of chunks in an SCTP packet that will be parsed for a packet that matches an existing association. This value is enforced to be greater or equal than **net.inet.ip.alias.sctp.initialising_chunk_proc_limit**. A high value is a DoS risk yet setting too low a value may result in important control chunks in the packet not being located and parsed.

net.inet.ip.alias.sctp.error_on_ootb: 1
Defines when the **nat** responds to any Out-of-the-Blue (OOTB) packets with ErrorM packets. An OOTB packet is a packet that arrives with no existing association registered in the **nat** and is not an INIT or ASCONF-AddIP packet:

0 ErrorM is never sent in response to OOTB packets.

1 ErrorM is only sent to OOTB packets received on the local side.

2 ErrorM is sent to the local side and on the global side ONLY if there is a partial match (ports and vtags match but the source global IP does not). This value is only useful if the **nat** is tracking global IP addresses.

3 ErrorM is sent in response to all OOTB packets on both the local and global side (DoS risk).

At the moment the default is 0, since the ErrorM packet is not yet supported by most SCTP stacks. When it is supported, and if not tracking global addresses, we recommend setting this

value to 1 to allow multi-homed local hosts to function with the **nat**. To track global addresses, we recommend setting this value to 2 to allow global hosts to be informed when they need to (re)send an ASCONF-AddIP. Value 3 should never be chosen (except for debugging) as the **nat** will respond to all OOTB global packets (a DoS risk).

net.inet.ip.alias.sctp.hashtable_size: 2003
: Size of hash tables used for **nat** lookups (100 < prime_number > 1000001). This value sets the **hash table** size for any future created **nat** instance and therefore must be set prior to creating a **nat** instance. The table sizes may be changed to suit specific needs. If there will be few concurrent associations, and memory is scarce, you may make these smaller. If there will be many thousands (or millions) of concurrent associations, you should make these larger. A prime number is best for the table size. The sysctl update function will adjust your input value to the next highest prime number.

net.inet.ip.alias.sctp.holddown_time: 0
: Hold association in table for this many seconds after receiving a SHUTDOWN-COMPLETE. This allows endpoints to correct shutdown gracefully if a shutdown_complete is lost and retransmissions are required.

net.inet.ip.alias.sctp.init_timer: 15
: Timeout value while waiting for (INIT-ACK|AddIP-ACK). This value cannot be 0.

net.inet.ip.alias.sctp.initialising_chunk_proc_limit: 2
: Defines the maximum number of chunks in an SCTP packet that will be parsed when no existing association exists that matches that packet. Ideally this packet will only be an INIT or ASCONF-AddIP packet. A higher value may become a DoS risk as malformed packets can consume processing resources.

net.inet.ip.alias.sctp.param_proc_limit: 25
: Defines the maximum number of parameters within a chunk that will be parsed in a packet. As for other similar sysctl variables, larger values pose a DoS risk.

net.inet.ip.alias.sctp.log_level: 0
: Level of detail in the system log messages (0 - minimal, 1 - event, 2 - info, 3 - detail, 4 - debug, 5 - max debug). May be a good option in high loss environments.

net.inet.ip.alias.sctp.shutdown_time: 15
: Timeout value while waiting for SHUTDOWN-COMPLETE. This value cannot be 0.

net.inet.ip.alias.sctp.track_global_addresses: 0

Enables/disables global IP address tracking within the **nat** and places an upper limit on the number of addresses tracked for each association:

0 Global tracking is disabled

>1 Enables tracking, the maximum number of addresses tracked for each association is limited to this value

This variable is fully dynamic, the new value will be adopted for all newly arriving associations, existing associations are treated as they were previously. Global tracking will decrease the number of collisions within the **nat** at a cost of increased processing load, memory usage, complexity, and possible **nat** state problems in complex networks with multiple **nats**. We recommend not tracking global IP addresses, this will still result in a fully functional **nat**.

net.inet.ip.alias.sctp.up_timer: 300
Timeout value to keep an association up with no traffic. This value cannot be 0.

net.inet.ip.dummynet.codel.interval: 100000
Default **codel** AQM interval in microseconds. The value must be in the range 1..5000000.

net.inet.ip.dummynet.codel.target: 5000
Default **codel** AQM target delay time in microseconds (the minimum acceptable persistent queue delay). The value must be in the range 1..5000000.

net.inet.ip.dummynet.expire: 1
Lazily delete dynamic pipes/queue once they have no pending traffic. You can disable this by setting the variable to 0, in which case the pipes/queues will only be deleted when the threshold is reached.

net.inet.ip.dummynet.fqcodel.flows: 1024
Defines the default total number of flow queues (sub-queues) that **fq_codel** creates and manages. The value must be in the range 1..65536.

net.inet.ip.dummynet.fqcodel.interval: 100000
Default **fq_codel** scheduler/AQM interval in microseconds. The value must be in the range 1..5000000.

net.inet.ip.dummynet.fqcodel.limit: 10240
The default hard size limit (in unit of packet) of all queues managed by an instance of the **fq_codel** scheduler. The value must be in the range 1..20480.

net.inet.ip.dummynet.fqcodel.quantum: 1514
The default quantum (credit) of the **fq_codel** in unit of byte. The value must be in the range 1..9000.

net.inet.ip.dummynet.fqcodel.target: 5000
Default **fq_codel** scheduler/AQM target delay time in microseconds (the minimum acceptable persistent queue delay). The value must be in the range 1..5000000.

net.inet.ip.dummynet.fqpie.alpha: 125
The default *alpha* parameter (scaled by 1000) for **fq_pie** scheduler/AQM. The value must be in the range 1..7000.

net.inet.ip.dummynet.fqpie.beta: 1250
The default *beta* parameter (scaled by 1000) for **fq_pie** scheduler/AQM. The value must be in the range 1..7000.

net.inet.ip.dummynet.fqpie.flows: 1024
Defines the default total number of flow queues (sub-queues) that **fq_pie** creates and manages. The value must be in the range 1..65536.

net.inet.ip.dummynet.fqpie.limit: 10240
The default hard size limit (in unit of packet) of all queues managed by an instance of the **fq_pie** scheduler. The value must be in the range 1..20480.

net.inet.ip.dummynet.fqpie.max_burst: 150000
The default maximum period of microseconds that **fq_pie** scheduler/AQM does not drop/mark packets. The value must be in the range 1..10000000.

net.inet.ip.dummynet.fqpie.max_ecnth: 99
The default maximum ECN probability threshold (scaled by 1000) for **fq_pie** scheduler/AQM. The value must be in the range 1..7000.

net.inet.ip.dummynet.fqpie.quantum: 1514
The default quantum (credit) of the **fq_pie** in unit of byte. The value must be in the range 1..9000.

net.inet.ip.dummynet.fqpie.target: 15000
The default **target** delay of the **fq_pie** in unit of microsecond. The value must be in the range 1..5000000.

net.inet.ip.dummynet.fqpie.tupdate: 15000
The default **tupdate** of the **fq_pie** in unit of microsecond. The value must be in the range 1..5000000.

net.inet.ip.dummynet.hash_size: 64
Default size of the hash table used for dynamic pipes/queues. This value is used when no **buckets** option is specified when configuring a pipe/queue.

net.inet.ip.dummynet.io_fast: 0
If set to a non-zero value, the "fast" mode of **dummynet** operation (see above) is enabled.

net.inet.ip.dummynet.io_pkt
Number of packets passed to **dummynet**.

net.inet.ip.dummynet.io_pkt_drop
Number of packets dropped by **dummynet**.

net.inet.ip.dummynet.io_pkt_fast
Number of packets bypassed by the **dummynet** scheduler.

net.inet.ip.dummynet.max_chain_len: 16
Target value for the maximum number of pipes/queues in a hash bucket. The product **max_chain_len*hash_size** is used to determine the threshold over which empty pipes/queues will be expired even when **net.inet.ip.dummynet.expire=0**.

net.inet.ip.dummynet.red_lookup_depth: 256

net.inet.ip.dummynet.red_avg_pkt_size: 512

net.inet.ip.dummynet.red_max_pkt_size: 1500
Parameters used in the computations of the drop probability for the RED algorithm.

net.inet.ip.dummynet.pie.alpha: 125
The default *alpha* parameter (scaled by 1000) for **pie** AQM. The value must be in the range 1..7000.

net.inet.ip.dummynet.pie.beta: 1250
The default *beta* parameter (scaled by 1000) for **pie** AQM. The value must be in the range 1..7000.

net.inet.ip.dummynet.pie.max_burst: 150000
The default maximum period of microseconds that **pie** AQM does not drop/mark packets. The value must be in the range 1..10000000.

net.inet.ip.dummynet.pie.max_ecnth: 99
The default maximum ECN probability threshold (scaled by 1000) for **pie** AQM. The value must be in the range 1..7000.

net.inet.ip.dummynet.pie.target: 15000
The default **target** delay of **pie** AQM in unit of microsecond. The value must be in the range 1..5000000.

net.inet.ip.dummynet.pie.tupdate: 15000
The default **tupdate** of **pie** AQM in unit of microsecond. The value must be in the range 1..5000000.

net.inet.ip.dummynet.pipe_byte_limit: 1048576

net.inet.ip.dummynet.pipe_slot_limit: 100
The maximum queue size that can be specified in bytes or packets. These limits prevent accidental exhaustion of resources such as mbufs. If you raise these limits, you should make sure the system is configured so that sufficient resources are available.

net.inet.ip.fw.autoinc_step: 100
Delta between rule numbers when auto-generating them. The value must be in the range 1..1000.

net.inet.ip.fw.curr_dyn_buckets: *net.inet.ip.fw.dyn_buckets*
The current number of buckets in the hash table for dynamic rules (readonly).

net.inet.ip.fw.debug: 1
Controls debugging messages produced by **ipfw**.

net.inet.ip.fw.default_rule: 65535
The default rule number (read-only). By the design of **ipfw**, the default rule is the last one, so its number can also serve as the highest number allowed for a rule.

net.inet.ip.fw.dyn_buckets: 256
The number of buckets in the hash table for dynamic rules. Must be a power of 2, up to 65536. It only takes effect when all dynamic rules have expired, so you are advised to use a **flush** command to make sure that the hash table is resized.

net.inet.ip.fw.dyn_count: 3
Current number of dynamic rules (read-only).

net.inet.ip.fw.dyn_keepalive: 1
Enables generation of keepalive packets for **keep-state** rules on TCP sessions. A keepalive is generated to both sides of the connection every 5 seconds for the last 20 seconds of the lifetime of the rule.

net.inet.ip.fw.dyn_max: 8192
Maximum number of dynamic rules. When you hit this limit, no more dynamic rules can be installed until old ones expire.

net.inet.ip.fw.dyn_ack_lifetime: 300

net.inet.ip.fw.dyn_syn_lifetime: 20

net.inet.ip.fw.dyn_fin_lifetime: 1

net.inet.ip.fw.dyn_rst_lifetime: 1

net.inet.ip.fw.dyn_udp_lifetime: 5

net.inet.ip.fw.dyn_short_lifetime: 30
These variables control the lifetime, in seconds, of dynamic rules. Upon the initial SYN exchange the lifetime is kept short, then increased after both SYN have been seen, then decreased again during the final FIN exchange or when a RST is received. Both *dyn_fin_lifetime* and *dyn_rst_lifetime* must be strictly lower than 5 seconds, the period of repetition of keepalives. The firewall enforces that.

net.inet.ip.fw.dyn_keep_states: 0
Keep dynamic states on rule/set deletion. States are relinked to default rule (65535). This can be handly for ruleset reload. Turned off by default.

net.inet.ip.fw.one_pass: 1
When set, the packet exiting from the **dummynet** pipe or from ng_ipfw(4) node is not passed though the firewall again. Otherwise, after an action, the packet is reinjected into the firewall at the next rule.

net.inet.ip.fw.tables_max: 128
Maximum number of tables.

net.inet.ip.fw.verbose: 1
Enables verbose messages.

net.inet.ip.fw.verbose_limit: 0
Limits the number of messages produced by a verbose firewall.

net.inet6.ip6.fw.deny_unknown_exthdrs: 1
If enabled packets with unknown IPv6 Extension Headers will be denied.

net.link.bridge.ipfw: 0
Controls whether bridged packets are passed to **ipfw**. Default is no.

net.inet.ip.fw.nat64_debug: 0
Controls debugging messages produced by **ipfw_nat64** module.

net.inet.ip.fw.nat64_direct_output: 0
Controls the output method used by **ipfw_nat64** module:

0 A packet is handled by **ipfw** twice. First time an original packet is handled by **ipfw** and consumed by **ipfw_nat64** translator. Then translated packet is queued via netisr to input processing again.

1 A packet is handled by **ipfw** only once, and after translation it will be pushed directly to outgoing interface.

INTERNAL DIAGNOSTICS

There are some commands that may be useful to understand current state of certain subsystems inside kernel module. These commands provide debugging output which may change without notice.

Currently the following commands are available as **internal** sub-options:

iflist Lists all interface which are currently tracked by **ipfw** with their in-kernel status.

talist List all table lookup algorithms currently available.

EXAMPLES

There are far too many possible uses of **ipfw** so this Section will only give a small set of examples.

BASIC PACKET FILTERING

This command adds an entry which denies all tcp packets from *cracker.evil.org* to the telnet port of

wolf.tambov.su from being forwarded by the host:

```
ipfw add deny tcp from cracker.evil.org to wolf.tambov.su telnet
```

This one disallows any connection from the entire cracker's network to my host:

```
ipfw add deny ip from 123.45.67.0/24 to my.host.org
```

A first and efficient way to limit access (not using dynamic rules) is the use of the following rules:

```
ipfw add allow tcp from any to any established
ipfw add allow tcp from net1 portlist1 to net2 portlist2 setup
ipfw add allow tcp from net3 portlist3 to net3 portlist3 setup
...
ipfw add deny tcp from any to any
```

The first rule will be a quick match for normal TCP packets, but it will not match the initial SYN packet, which will be matched by the **setup** rules only for selected source/destination pairs. All other SYN packets will be rejected by the final **deny** rule.

If you administer one or more subnets, you can take advantage of the address sets and or-blocks and write extremely compact rulesets which selectively enable services to blocks of clients, as below:

```
goodguys="{ 10.1.2.0/24{20,35,66,18} or 10.2.3.0/28{6,3,11} }"
badguys="10.1.2.0/24{8,38,60}"
ipfw add allow ip from ${goodguys} to any
ipfw add deny ip from ${badguys} to any
... normal policies ...
```

Allow any transit packets coming from single vlan 10 and going out to vlans 100-1000:

```
ipfw add 10 allow out recv vlan10 \
{ xmit vlan1000 or xmit "vlan[1-9]??" }
```

The **verrevpath** option could be used to do automated anti-spoofing by adding the following to the top of a ruleset:

```
ipfw add deny ip from any to any not verrevpath in
```

This rule drops all incoming packets that appear to be coming to the system on the wrong interface. For

example, a packet with a source address belonging to a host on a protected internal network would be dropped if it tried to enter the system from an external interface.

The **antispoof** option could be used to do similar but more restricted anti-spoofing by adding the following to the top of a ruleset:

```
ipfw add deny ip from any to any not antispoof in
```

This rule drops all incoming packets that appear to be coming from another directly connected system but on the wrong interface. For example, a packet with a source address of 192.168.0.0/24, configured on fxp0, but coming in on fxp1 would be dropped.

The **setdscp** option could be used to (re)mark user traffic, by adding the following to the appropriate place in ruleset:

```
ipfw add setdscp be ip from any to any dscp af11,af21
```

SELECTIVE MIRRORING

If your network has network traffic analyzer connected to your host directly via dedicated interface or remotely via RSPAN vlan, you can selectively mirror some Ethernet layer2 frames to the analyzer.

First, make sure your firewall is already configured and runs. Then, enable layer2 processing if not already enabled:

```
sysctl net.link.ether.ipfw=1
```

Next, load needed additional kernel modules:

```
kldload ng_ether ng_ipfw
```

Optionally, make system load these modules automatically at startup:

```
sysrc kld_list+="ng_ether ng_ipfw"
```

Next, configure ng_ipfw(4) kernel module to transmit mirrored copies of layer2 frames out via vlan900 interface:

```
ngctl connect ipfw: vlan900: 1 lower
```

Think of "1" here as of "mirroring instance index" and vlan900 is its destination. You can have arbitrary

number of instances. Refer to ng_ipfw(4) for details.

At last, actually start mirroring of selected frames using "instance 1". For frames incoming from em0 interface:

```
ipfw add ngtee 1 ip from any to 192.168.0.1 layer2 in recv em0
```

For frames outgoing to em0 interface:

```
ipfw add ngtee 1 ip from any to 192.168.0.1 layer2 out xmit em0
```

For both incoming and outgoing frames while flowing through em0:

```
ipfw add ngtee 1 ip from any to 192.168.0.1 layer2 via em0
```

Make sure you do not perform mirroring for already duplicated frames or kernel may hang as there is no safety net.

DYNAMIC RULES

In order to protect a site from flood attacks involving fake TCP packets, it is safer to use dynamic rules:

```
ipfw add check-state
ipfw add deny tcp from any to any established
ipfw add allow tcp from my-net to any setup keep-state
```

This will let the firewall install dynamic rules only for those connection which start with a regular SYN packet coming from the inside of our network. Dynamic rules are checked when encountering the first occurrence of a **check-state**, **keep-state** or **limit** rule. A **check-state** rule should usually be placed near the beginning of the ruleset to minimize the amount of work scanning the ruleset. Your mileage may vary.

For more complex scenarios with dynamic rules **record-state** and **defer-action** can be used to precisely control creation and checking of dynamic rules. Example of usage of these options are provided in *NETWORK ADDRESS TRANSLATION (NAT)* Section.

To limit the number of connections a user can open you can use the following type of rules:

```
ipfw add allow tcp from my-net/24 to any setup limit src-addr 10
ipfw add allow tcp from any to me setup limit src-addr 4
```

The former (assuming it runs on a gateway) will allow each host on a /24 network to open at most 10 TCP connections. The latter can be placed on a server to make sure that a single client does not use more than 4 simultaneous connections.

BEWARE: stateful rules can be subject to denial-of-service attacks by a SYN-flood which opens a huge number of dynamic rules. The effects of such attacks can be partially limited by acting on a set of sysctl(8) variables which control the operation of the firewall.

Here is a good usage of the **list** command to see accounting records and timestamp information:

```
ipfw -at list
```

or in short form without timestamps:

```
ipfw -a list
```

which is equivalent to:

```
ipfw show
```

Next rule diverts all incoming packets from 192.168.2.0/24 to divert port 5000:

```
ipfw divert 5000 ip from 192.168.2.0/24 to any in
```

TRAFFIC SHAPING

The following rules show some of the applications of **ipfw** and **dummynet** for simulations and the like.

This rule drops random incoming packets with a probability of 5%:

```
ipfw add prob 0.05 deny ip from any to any in
```

A similar effect can be achieved making use of **dummynet** pipes:

```
dnctl add pipe 10 ip from any to any
dnctl pipe 10 config plr 0.05
```

We can use pipes to artificially limit bandwidth, e.g. on a machine acting as a router, if we want to limit traffic from local clients on 192.168.2.0/24 we do:

```
ipfw add pipe 1 ip from 192.168.2.0/24 to any out
```

```
dnctl pipe 1 config bw 300Kbit/s queue 50KBytes
```

note that we use the **out** modifier so that the rule is not used twice. Remember in fact that **ipfw** rules are checked both on incoming and outgoing packets.

Should we want to simulate a bidirectional link with bandwidth limitations, the correct way is the following:

```
ipfw add pipe 1 ip from any to any out
ipfw add pipe 2 ip from any to any in
dnctl pipe 1 config bw 64Kbit/s queue 10Kbytes
dnctl pipe 2 config bw 64Kbit/s queue 10Kbytes
```

The above can be very useful, e.g. if you want to see how your fancy Web page will look for a residential user who is connected only through a slow link. You should not use only one pipe for both directions, unless you want to simulate a half-duplex medium (e.g. AppleTalk, Ethernet, IRDA). It is not necessary that both pipes have the same configuration, so we can also simulate asymmetric links.

Should we want to verify network performance with the RED queue management algorithm:

```
ipfw add pipe 1 ip from any to any
dnctl pipe 1 config bw 500Kbit/s queue 100 red 0.002/30/80/0.1
```

Another typical application of the traffic shaper is to introduce some delay in the communication. This can significantly affect applications which do a lot of Remote Procedure Calls, and where the round-trip-time of the connection often becomes a limiting factor much more than bandwidth:

```
ipfw add pipe 1 ip from any to any out
ipfw add pipe 2 ip from any to any in
dnctl pipe 1 config delay 250ms bw 1Mbit/s
dnctl pipe 2 config delay 250ms bw 1Mbit/s
```

Per-flow queueing can be useful for a variety of purposes. A very simple one is counting traffic:

```
ipfw add pipe 1 tcp from any to any
ipfw add pipe 1 udp from any to any
ipfw add pipe 1 ip from any to any
dnctl pipe 1 config mask all
```

The above set of rules will create queues (and collect statistics) for all traffic. Because the pipes have no

limitations, the only effect is collecting statistics. Note that we need 3 rules, not just the last one, because when **ipfw** tries to match IP packets it will not consider ports, so we would not see connections on separate ports as different ones.

A more sophisticated example is limiting the outbound traffic on a net with per-host limits, rather than per-network limits:

```
ipfw add pipe 1 ip from 192.168.2.0/24 to any out
ipfw add pipe 2 ip from any to 192.168.2.0/24 in
dnctl pipe 1 config mask src-ip 0x000000ff bw 200Kbit/s queue 20Kbytes
dnctl pipe 2 config mask dst-ip 0x000000ff bw 200Kbit/s queue 20Kbytes
```

LOOKUP TABLES

In the following example, we need to create several traffic bandwidth classes and we need different hosts/networks to fall into different classes. We create one pipe for each class and configure them accordingly. Then we create a single table and fill it with IP subnets and addresses. For each subnet/host we set the argument equal to the number of the pipe that it should use. Then we classify traffic using a single rule:

```
dnctl pipe 1 config bw 1000Kbyte/s
dnctl pipe 4 config bw 4000Kbyte/s
...
ipfw table T1 create type addr
ipfw table T1 add 192.168.2.0/24 1
ipfw table T1 add 192.168.0.0/27 4
ipfw table T1 add 192.168.0.2 1
...
ipfw add pipe tablearg ip from 'table(T1)' to any
```

Using the **fwd** action, the table entries may include hostnames and IP addresses.

```
ipfw table T2 create type addr valtype ipv4
ipfw table T2 add 192.168.2.0/24 10.23.2.1
ipfw table T2 add 192.168.0.0/27 router1.dmz
...
ipfw add 100 fwd tablearg ip from any to 'table(T2)'
```

In the following example per-interface firewall is created:

```
ipfw table IN create type iface valtype skipto,fib
```

```
ipfw table IN add vlan20 12000,12
ipfw table IN add vlan30 13000,13
ipfw table OUT create type iface valtype skipto
ipfw table OUT add vlan20 22000
ipfw table OUT add vlan30 23000
..
ipfw add 100 setfib tablearg ip from any to any recv 'table(IN)' in
ipfw add 200 skipto tablearg ip from any to any recv 'table(IN)' in
ipfw add 300 skipto tablearg ip from any to any xmit 'table(OUT)' out
```

The following example illustrate usage of flow tables:

```
ipfw table fl create type flow:src-ip,proto,dst-ip,dst-port
ipfw table fl add 2a02:6b8:77::88,tcp,2a02:6b8:77::99,80 11
ipfw table fl add 10.0.0.1,udp,10.0.0.2,53 12
..
ipfw add 100 allow ip from any to any flow 'table(fl,11)' recv ix0
```

SETS OF RULES

To add a set of rules atomically, e.g. set 18:

```
ipfw set disable 18
ipfw add NN set 18 ...        # repeat as needed
ipfw set enable 18
```

To delete a set of rules atomically the command is simply:

```
ipfw delete set 18
```

To test a ruleset and disable it and regain control if something goes wrong:

```
ipfw set disable 18
ipfw add NN set 18 ...        # repeat as needed
ipfw set enable 18; echo done; sleep 30 && ipfw set disable 18
```

Here if everything goes well, you press control-C before the "sleep" terminates, and your ruleset will be left active. Otherwise, e.g. if you cannot access your box, the ruleset will be disabled after the sleep terminates thus restoring the previous situation.

To show rules of the specific set:

ipfw set 18 show

To show rules of the disabled set:

ipfw -S set 18 show

To clear a specific rule counters of the specific set:

ipfw set 18 zero NN

To delete a specific rule of the specific set:

ipfw set 18 delete NN

NAT, REDIRECT AND LSNAT

First redirect all the traffic to nat instance 123:

ipfw add nat 123 all from any to any

Then to configure nat instance 123 to alias all the outgoing traffic with ip 192.168.0.123, blocking all incoming connections, trying to keep same ports on both sides, clearing aliasing table on address change and keeping a log of traffic/link statistics:

ipfw nat 123 config ip 192.168.0.123 log deny_in reset same_ports

Or to change address of instance 123, aliasing table will be cleared (see reset option):

ipfw nat 123 config ip 10.0.0.1

To see configuration of nat instance 123:

ipfw nat 123 show config

To show logs of all instances:

ipfw nat show log

To see configurations of all instances:

ipfw nat show config

Or a redirect rule with mixed modes could looks like:

```
ipfw nat 123 config redirect_addr 10.0.0.1 10.0.0.66
                    redirect_port tcp 192.168.0.1:80 500
                    redirect_proto udp 192.168.1.43 192.168.1.1
                    redirect_addr 192.168.0.10,192.168.0.11
                                10.0.0.100      # LSNAT
                    redirect_port tcp 192.168.0.1:80,192.168.0.10:22
                                500             # LSNAT
```

or it could be split in:

```
ipfw nat 1 config redirect_addr 10.0.0.1 10.0.0.66
ipfw nat 2 config redirect_port tcp 192.168.0.1:80 500
ipfw nat 3 config redirect_proto udp 192.168.1.43 192.168.1.1
ipfw nat 4 config redirect_addr 192.168.0.10,192.168.0.11,192.168.0.12
                                10.0.0.100
ipfw nat 5 config redirect_port tcp
                    192.168.0.1:80,192.168.0.10:22,192.168.0.20:25 500
```

Sometimes you may want to mix NAT and dynamic rules. It could be achieved with **record-state** and **defer-action** options. Problem is, you need to create dynamic rule before NAT and check it after NAT actions (or vice versa) to have consistent addresses and ports. Rule with **keep-state** option will trigger activation of existing dynamic state, and action of such rule will be performed as soon as rule is matched. In case of NAT and **allow** rule packet need to be passed to NAT, not allowed as soon is possible.

There is example of set of rules to achieve this. Bear in mind that this is example only and it is not very useful by itself.

On way out, after all checks place this rules:

```
ipfw add allow record-state defer-action
ipfw add nat 1
```

And on way in there should be something like this:

```
ipfw add nat 1
ipfw add check-state
```

Please note, that first rule on way out doesn't allow packet and doesn't execute existing dynamic rules. All it does, create new dynamic rule with **allow** action, if it is not created yet. Later, this dynamic rule is used on way in by **check-state** rule.

CONFIGURING CODEL, PIE, FQ-CODEL and FQ-PIE AQM

codel and **pie** AQM can be configured for **dummynet pipe** or **queue**.

To configure a **pipe** with **codel** AQM using default configuration for traffic from 192.168.0.0/24 and 1Mbits/s rate limit, we do:

```
dnctl pipe 1 config bw 1mbits/s codel
ipfw add 100 pipe 1 ip from 192.168.0.0/24 to any
```

To configure a **queue** with **codel** AQM using different configurations parameters for traffic from 192.168.0.0/24 and 1Mbits/s rate limit, we do:

```
dnctl pipe 1 config bw 1mbits/s
dnctl queue 1 config pipe 1 codel target 8ms interval 160ms ecn
ipfw add 100 queue 1 ip from 192.168.0.0/24 to any
```

To configure a **pipe** with **pie** AQM using default configuration for traffic from 192.168.0.0/24 and 1Mbits/s rate limit, we do:

```
dnctl pipe 1 config bw 1mbits/s pie
ipfw add 100 pipe 1 ip from 192.168.0.0/24 to any
```

To configure a **queue** with **pie** AQM using different configuration parameters for traffic from 192.168.0.0/24 and 1Mbits/s rate limit, we do:

```
dnctl pipe 1 config bw 1mbits/s
dnctl queue 1 config pipe 1 pie target 20ms tupdate 30ms ecn
ipfw add 100 queue 1 ip from 192.168.0.0/24 to any
```

fq_codel and **fq_pie** AQM can be configured for **dummynet** schedulers.

To configure **fq_codel** scheduler using different configurations parameters for traffic from 192.168.0.0/24 and 1Mbits/s rate limit, we do:

```
dnctl pipe 1 config bw 1mbits/s
dnctl sched 1 config pipe 1 type fq_codel
```

```
dnctl queue 1 config sched 1
ipfw add 100 queue 1 ip from 192.168.0.0/24 to any
```

To change **fq_codel** default configuration for a **sched** such as disable ECN and change the *target* to 10ms, we do:

```
dnctl sched 1 config pipe 1 type fq_codel target 10ms noecn
```

Similar to **fq_codel**, to configure **fq_pie** scheduler using different configurations parameters for traffic from 192.168.0.0/24 and 1Mbits/s rate limit, we do:

```
dnctl pipe 1 config bw 1mbits/s
dnctl sched 1 config pipe 1 type fq_pie
dnctl queue 1 config sched 1
ipfw add 100 queue 1 ip from 192.168.0.0/24 to any
```

The configurations of **fq_pie sched** can be changed in a similar way as for **fq_codel**

SEE ALSO

cpp(1), m4(1), fnmatch(3), altq(4), divert(4), dummynet(4), if_bridge(4), ip(4), ipfirewall(4), ng_ether(4), ng_ipfw(4), protocols(5), services(5), init(8), kldload(8), reboot(8), sysctl(8), syslogd(8), sysrc(8)

HISTORY

The **ipfw** utility first appeared in FreeBSD 2.0. **dummynet** was introduced in FreeBSD 2.2.8. Stateful extensions were introduced in FreeBSD 4.0. **ipfw2** was introduced in Summer 2002.

AUTHORS

Ugen J. S. Antsilevich,
Poul-Henning Kamp,
Alex Nash,
Archie Cobbs,
Luigi Rizzo,
Rasool Al-Saadi.

API based upon code written by Daniel Boulet for BSDI.

Dummynet has been introduced by Luigi Rizzo in 1997-1998.

Some early work (1999-2000) on the **dummynet** traffic shaper supported by Akamba Corp.

The ipfw core (ipfw2) has been completely redesigned and reimplemented by Luigi Rizzo in summer 2002. Further actions and options have been added by various developers over the years.

In-kernel NAT support written by Paolo Pisati <*piso@FreeBSD.org*> as part of a Summer of Code 2005 project.

SCTP **nat** support has been developed by The Centre for Advanced Internet Architectures (CAIA) <http://www.caia.swin.edu.au>. The primary developers and maintainers are David Hayes and Jason But. For further information visit: <http://www.caia.swin.edu.au/urp/SONATA>

Delay profiles have been developed by Alessandro Cerri and Luigi Rizzo, supported by the European Commission within Projects Onelab and Onelab2.

CoDel, PIE, FQ-CoDel and FQ-PIE AQM for Dummynet have been implemented by The Centre for Advanced Internet Architectures (CAIA) in 2016, supported by The Comcast Innovation Fund. The primary developer is Rasool Al-Saadi.

BUGS

The syntax has grown over the years and sometimes it might be confusing. Unfortunately, backward compatibility prevents cleaning up mistakes made in the definition of the syntax.

!!! WARNING !!!

Misconfiguring the firewall can put your computer in an unusable state, possibly shutting down network services and requiring console access to regain control of it.

Incoming packet fragments diverted by **divert** are reassembled before delivery to the socket. The action used on those packet is the one from the rule which matches the first fragment of the packet.

Packets diverted to userland, and then reinserted by a userland process may lose various packet attributes. The packet source interface name will be preserved if it is shorter than 8 bytes and the userland process saves and reuses the sockaddr_in (as does natd(8)); otherwise, it may be lost. If a packet is reinserted in this manner, later rules may be incorrectly applied, making the order of **divert** rules in the rule sequence very important.

Dummynet drops all packets with IPv6 link-local addresses.

Rules using **uid** or **gid** may not behave as expected. In particular, incoming SYN packets may have no uid or gid associated with them since they do not yet belong to a TCP connection, and the uid/gid associated with a packet may not be as expected if the associated process calls setuid(2) or similar

system calls.

Rule syntax is subject to the command line environment and some patterns may need to be escaped with the backslash character or quoted appropriately.

Due to the architecture of libalias(3), ipfw nat is not compatible with the TCP segmentation offloading (TSO). Thus, to reliably nat your network traffic, please disable TSO on your NICs using ifconfig(8).

ICMP error messages are not implicitly matched by dynamic rules for the respective conversations. To avoid failures of network error detection and path MTU discovery, ICMP error messages may need to be allowed explicitly through static rules.

Rules using **call** and **return** actions may lead to confusing behaviour if ruleset has mistakes, and/or interaction with other subsystems (netgraph, dummynet, etc.) is used. One possible case for this is packet leaving **ipfw** in subroutine on the input pass, while later on output encountering unpaired **return** first. As the call stack is kept intact after input pass, packet will suddenly return to the rule number used on input pass, not on output one. Order of processing should be checked carefully to avoid such mistakes.