Design and Performance of the OpenBSD Stateful Packet Filter (pf)

Daniel Hartmeier[*]
Systor AG
dhartmei@openbsd.org

Abstract

With more and more hosts being connected to the Internet, the importance of securing connected networks has increased, too. One mechanism to provide enhanced security for a network is to filter out potentially malicious network packets. Firewalls are designed to provide "policy-based" network filtering.

A firewall may consist of several components. Its key component is usually a packet filter. The packet filter may be stateful to reach more informed decisions. The state allows the packet filter to keep track of established connections so that arriving packets could be associated with them. On the other hand, a stateless packet filter bases its decisions solely on individual packets. With release 3.0, OpenBSD includes a new Stateful Packet Filter (*pf*) in the base install. *pf* implements traditional packet filtering with some additional novel algorithms. This paper describes the design and implementation of *pf* and compares its scalability and performance with existing packet filter implementations.

1 Introduction

The main emphasis of the OpenBSD project is proactive and effective computer security. The integration of a Stateful Packet Filter is an important aspect. Since 1996, Darren Reed's *IPFilter* has been included in the OpenBSD base install. It was removed after its license turned out to be incompatible with OpenBSD's goal of providing software that is free to use, modify and redistribute in any way for everyone.

[*]Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0537.

While an acceptable Open Source license was a prerequisite for any replacement, we used this opportunity to develop a new packet filter that employed optimized data structures and algorithms to achieve good performance for stateful filtering and address translation. The resulting code base is small and thus facilitates future extensions.

The remainder of this paper is organized as follows. Section 2 outlines the design of the packet filter. In Section 3 we compare *pf*'s performance with other packet filters and discuss the results. Section 4 presents future work. Finally, we conclude in Section 5.

2 Design

The Stateful Packet Filter resides in the kernel and inspects every IP packet that enters or leaves the stack. It may reach one of several decisions:

- to pass a packet unmodified or modified,
- to silently block a packet, or
- to reject packet with a response, e.g., sending a TCP reset.

The filter itself consists of two basic elements, the filter rules and the state table.

2.1 Filter rules

Every packet is compared against the filter rule set. The rule set consists of a linked list of rules. Each rule contains a set of parameters that determines the set of packets the rule applies to. The parameters may be the source or destination address, the protocol, port numbers, etc. For a packet that matches the rule, the specified *pass* or *block* action is taken. *Block* means that the packet is dropped by the filter, and *pass* means that the packet is forwarded to its destination.

During rule set evaluation, the packet is compared against all rules from the beginning to the end. A packet can match more than one rule, in which case the last matching rule is used. This mechanism allows overriding general rules with more specific rules, like blocking all packets first and then passing specific packets. The last matching rule determines if the packet is passed or blocked according to its action field.

A matching rule that has been flagged as *final* terminates rule evaluation for the current packet. The action from that rule is applied to the packet. This prevents *final* rules from being overridden by subsequent rules.

The rule set is organized in a multiple linked list. This allows *pf* to perform automatic optimization of the rule set as discussed in Section 2.8.

2.2 State table

Stateful packet filtering implies that a firewall inspects not only single packets, but also that it knows about established connections. Any rule that passes a packet may create an entry in the state table. Before the filter rule set is evaluated for a packet, the state table is searched for a matching entry. If a packet is part of a tracked connection, it is passed unconditionally, without rule set evaluation.

For TCP, state matching involves checking sequence numbers against expected windows [8], which improves security against sequence number attacks.

UDP is stateless by nature: packets are considered to be part of the same connection if the host addresses and ports match a state entry. UDP state entries have an adaptive expiration scheme. The first UDP packet could either be a one-shot packet or the beginning of a UDP pseudo-connection. The first packet will create a state entry with a low timeout. If the other endpoint responds, *pf* will consider it a pseudo-connection with bidirectional communication and allow more flexibility in the duration of the state entry.

ICMP packets fall into two categories: ICMP error messages which refer to other packets, and ICMP queries and replies which are handled separately. If an ICMP error message corresponds to a connection in the state table, it is passed. ICMP queries and replies create their own state, similar to UDP states. As an example, an ICMP echo reply matches the state an ICMP echo request created. This is necessary so that applications like *ping* or *traceroute* work across a Stateful Packet Filter.

pf stores state entries in an AVL tree. An AVL tree is a balanced binary search tree. This container provides efficient search functionality which scales well for large trees. It guarantees the same $O(\log n)$ behavior even in worst case. Although alternative containers like hash tables allow searches in constant time, they also have their drawbacks. Hash tables have a fixed size by nature. As the number of entries grows, collisions occur (if two entries have the same hash value) and several entries end up in the same hash bucket. An attacker can trigger the worst case behavior by opening connections that lead to hash collisions and state entries in the same hash bucket. To accommodate this case, the hash buckets themselves would have to be binary search trees, otherwise the worst case behavior allows for denial of service attacks. The hash table would therefore only cover a shallow part of the entire search tree, and reduce only the first few levels of binary search, at considerable memory cost.

2.3 Network address translation

Network address translation (NAT) is commonly used to allow hosts with IP addresses from a private network to share an Internet connection using a single route-able address. A NAT gateway replaces the IP addresses and ports from packet that traverse the gateway with its own address information. Performing NAT in a Stateful Packet Filter is a natural extension, and *pf* combines NAT mappings and state table entries. This is a key design decision which has proved itself valuable.

The state table contains entries with three address/port pairs: the internal, the gateway and the external pair. Two trees contain the keys, one sorted on internal and external pair, the other sorted on external and gateway pair. This allows to find not only a matching state for outgoing and incoming packets, but also provides the NAT mapping in the same step without additional lookup costs. *pf* also supports port redirection and bidirectional translation. Application-level proxies reside in userland, e.g., ftp-proxy which allows active mode FTP for clients behind a NAT gateway.

2.4 Normalization

IP normalization removes interpretation ambiguities from IP traffic [5]. For example, operating systems resolve overlapping IP fragments in different ways. Some keep the old data while others replace the old data with data from a newly arrived fragment. For systems that resolve overlaps in favor of new fragments, it is possible to rewrite the protocol headers after they have been inspected by the firewall.

While OpenBSD itself is not vulnerable to fragmentation attacks [3, 4], it protects machines with less secure stacks behind it. Fragments are cached and reassembled by *pf* so that any conflict between overlapping fragments is resolved before a packet is received by its destination [2].

2.5 Sequence number modulation

pf can modulate TCP sequence numbers by adding a random number to all sequence numbers of a connection. This protects hosts with weak sequence number generators from attacks.

2.6 Logging

pf logs packets via bpf [6] from a virtual network interface called *pflog0*. This allows all of the existing network monitoring applications to monitor the *pf* logs with minimal modifications. *tcpdump* can even be used to monitor the logging device in real time and apply arbitrary filters to display only the applicable packets.

2.7 States vs. rule evaluation

Rule set evaluation scales with $O(n)$, where n is the number of rules in the set. However, state lookup scales with $O(\log m)$, where m is the number of states. The constant cost of one state key comparison is not significantly higher than the comparison of the parameters of one rule. This means that even with smaller rule sets, filtering statefully is actually more efficient as packets that match an entry in the state table are not evaluated by the rule set.

2.8 Transparent rule set evaluation optimization

pf automatically optimizes the evaluation of the rule set. If a group of consecutive rules all contain the same parameter, e.g., "source address equals 10.1.2.3," and a packet does not match this parameter when the first rule of the group is evaluated, the whole group of rules is skipped, as the packet can not possibly match any of the rules in the group.

When a rule set is loaded, the kernel traverses the set to calculate these so-called *skip-steps*. In each rule, for each parameter, there is a pointer set to the next rule that specifies a different value for the parameter.

During rule set evaluation, if a packet does not match a rule parameter, the pointer is used to skip to the next rule which could match, instead of the very next rule in the set.

The *skip-steps* optimization is completely transparent to the user because it happens automatically without changing the meaning of any rule set.

The performance gain depends on the specific rule set. In the worst case, all *skip-steps* have a length of one so that no rules can be skipped during evaluation, because the parameters of consecutive rules are always different. However, even in the worst case, performance does not decrease compared to an unoptimized version.

An average rule set, however, results in larger *skip-steps* which are responsible for a significant performance gain. The cost of evaluating a rule set is often reduced by an order of magnitude, as only every 10th to 100th rule is actually evaluated.

Firewall administrators can increase the likelihood of *skip-steps* optimizations and thereby improving the performance of rule evaluation by sorting blocks of rules on parameters in a documented order.

Automatically generated groups of rules are already sorted in optimal order, e.g., this happens when one rule contains parameter lists that are expanded internally to several new rules.

3 Performance evaluation

We evaluate the performance of the packet filter by using two hosts with two network interface cards each, connected with two crossover Cat5 cables, in 10baseT unidirectional mode.

The *tester* host uses a libnet program to generate TCP packets as ethernet frames. They are sent through the first interface to the *firewall* host and captured as ethernet frames on the second interface of the *tester* using libpcap. The two hosts do not have any other network connections.

The firewall is configured to forward IP packets between its interfaces, so that the packets sent by the tester are forwarded through the other interface back to the tester.

The firewall is an i386 machine with a Pentium 166 MHz CPU and 64 MB RAM; the tester is a faster i386 machine. All four network interface cards are identical NetGear PCI cards, using the sis driver.

Arp table entries are static, and the only packets traversing the wires are the packets generated by the tester and forwarded back by the firewall.

The generated packets contain time stamps and serial numbers, that allow the tester to determine latency and packet loss rate.

The tester is sending packets of defined size (bytes/packet, including ethernet header and checksum) at defined rates (packets/s), and measures the rate of received packets, the average latency and loss rate (percentage of packets lost). For each combination, the measuring period is at least ten seconds. Latency is the average for all packets returned to the tester during the measuring period. Lost packets are not counted towards latency.

Successively, the following tests are repeated with the same firewall running OpenBSD 3.0 with *pf*, OpenBSD 3.0 with *IPFilter* and GNU/Linux RedHat 7.2 with *iptables*.

3.1 Unfiltered

The first test is run without interposing the packet filter into the network stack on the firewall.

Figure 1 shows that both the tester and the firewall are able to handle packets at the maximum frame rate [1] for all packet sizes of 128 bytes and above. All further tests are done using packet sizes of either 128 or 256 bytes. The degradation for the GNU/Linux machine seems slightly worse than for the OpenBSD machine.

3.2 Stateless filtering with increasing rule set size

In the second test, the packet filter is enabled and the size of the filter rule set is increased repeatedly. The rules are chosen so that the packet filter is forced to evaluate the entire rule set and then pass each packet without creating state table entries. The generated packets contain random port numbers to defeat any mechanisms used by the packet filter to cache rule set evaluations.

Figures 2, 3 and 4 show throughput, latency and loss depending on sending rate, for a set of 100 rules, using 256 byte packets. *Iptables* outperforms both *pf* and *IPFilter* in this test. It has a higher maximum throughput and lower latency compared to the other two packet filters.

Each packet filter has a distinct maximum lossless throughput. If the sending rate exceeds this maximum, latency increases quickly and packet loss occurs. For all three filters, latency is nearly identical below the maximum lossless rate. When the sending rate is increased beyond the point where loss occurs, throughput actually decreases. In such overloaded condition, all three packet filter consume all CPU resources and the console becomes unresponsive. After the sending rate is lowered below the maximum throughput rate, each one of them recovers quickly.

The test is repeated with various rule set sizes between one and ten thousand. For each rule set size, the maximum throughput rate possible without packet loss is noted. The results are shown in

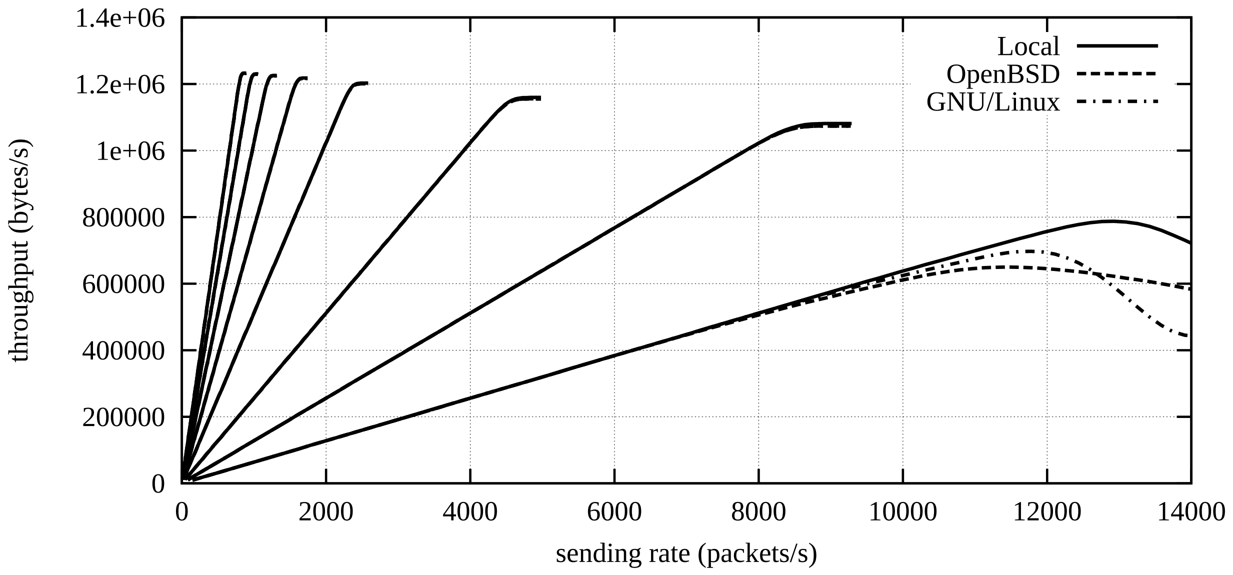

Figure 1: Unfiltered

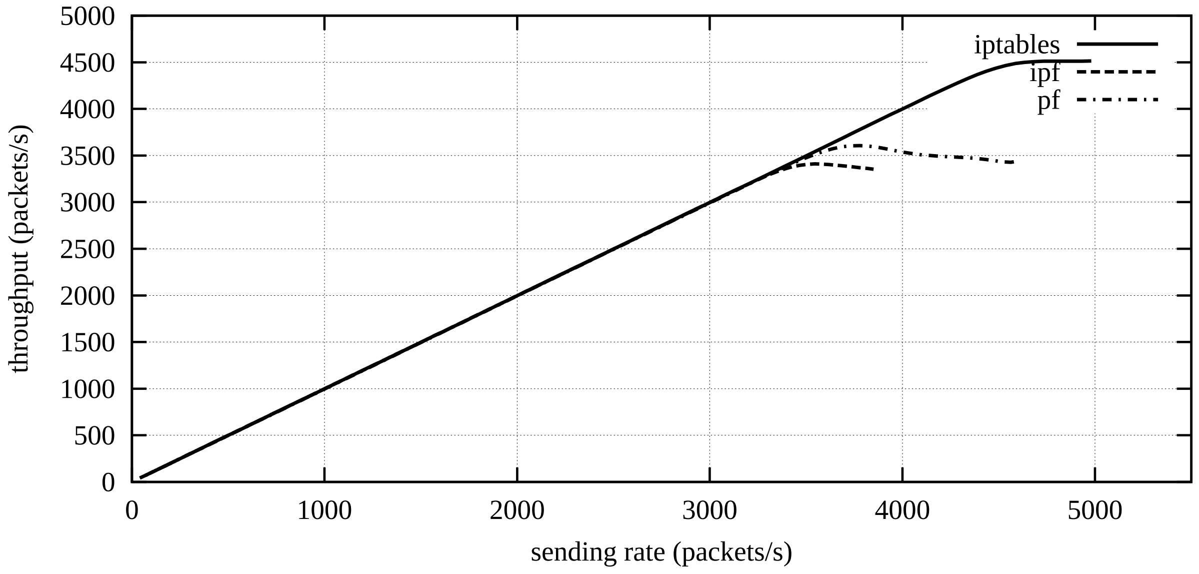

Figure 2: Stateless filtering with 100 rules (throughput)

Figure 5 as a function of the rule set size. Both *pf* and *IPFilter* evaluate the rule set twice for each packet, once incoming on the first interface and once outgoing on the second interface. *Iptables'* performance advantage is due to the fact that it evaluates the rule set only once by using a forwarding chain. The forwarding chain evaluates the rules set based on a packet's complete path through the machine.

3.3 Stateful filtering with increasing state table size

The third test uses a rule set that contains only a single rule to pass all packets statefully, *i.e.*, new state is created for packets that do not belong to any connection tracked by the existing state. To prime the state table, the tester establishes a de-

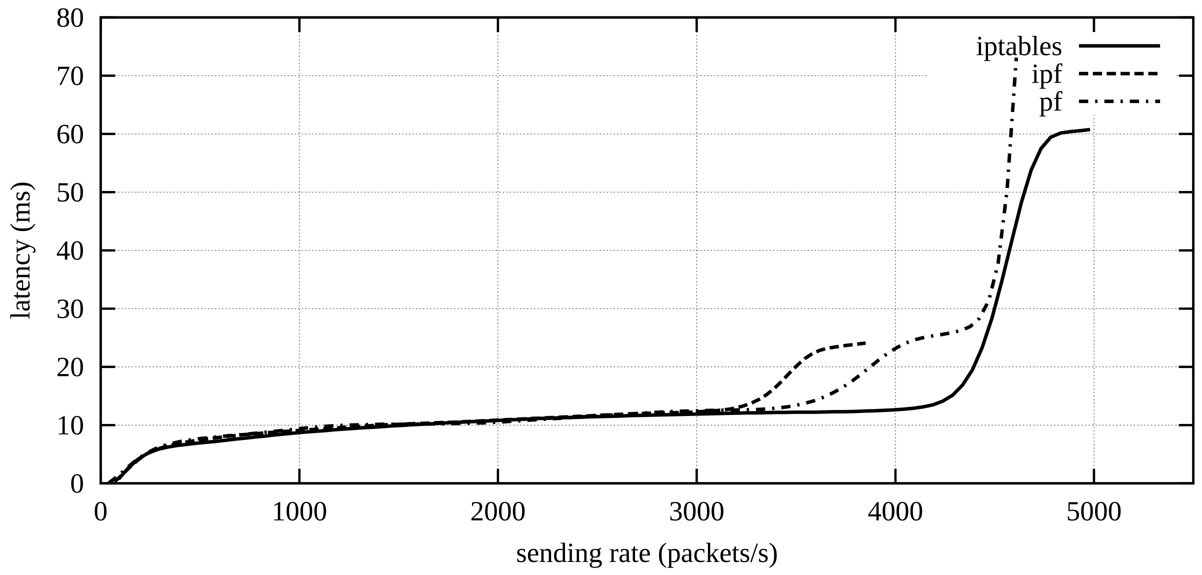

Figure 3: Stateless filtering with 100 rules (latency)

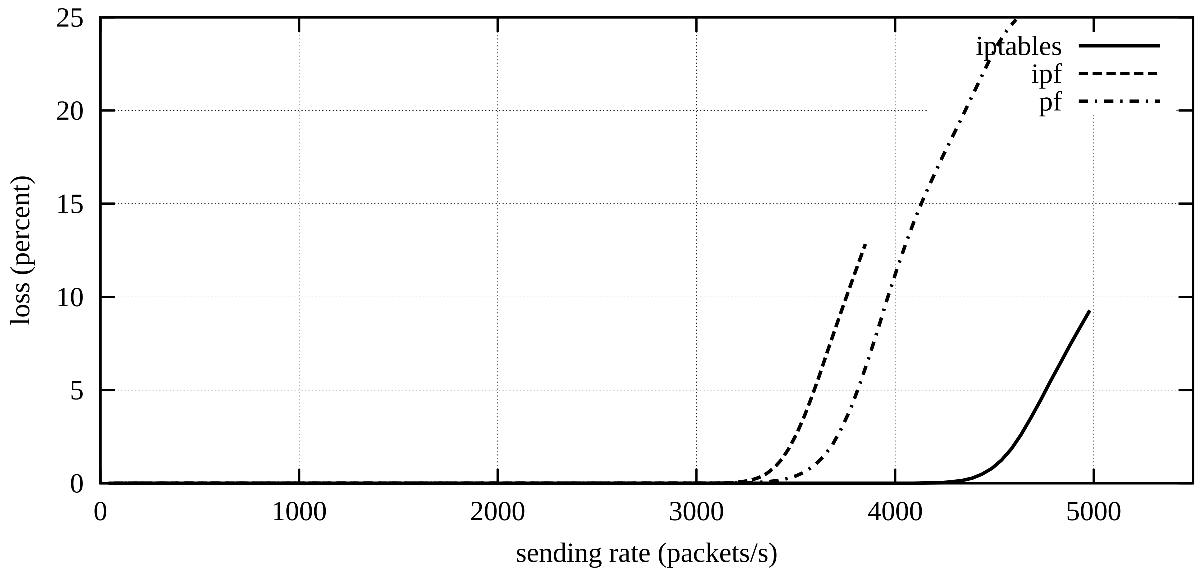

Figure 4: Stateless filtering with 100 rules (loss)

fined number of connections with itself by completing TCP handshakes. After all connections are established, the tester sends random packets matching the established connections with uniform distribution. During the entire test, all state table entries are used, no entries time out and no new entries are added. *Iptables* is not included in the stateful tests as it does not perform proper state tracking as explained below.

Figure 6 compares the throughput in relation to the sending rate for stateful filtering with twenty thousand state entries. Both *pf* and *IPFilter* exhibit the same behavior when overloaded. However, *IPFilter* reaches overload at a packet rate of about four thousand packets per second whereas, *pf* does not reach overload until about six thousand packets per second.

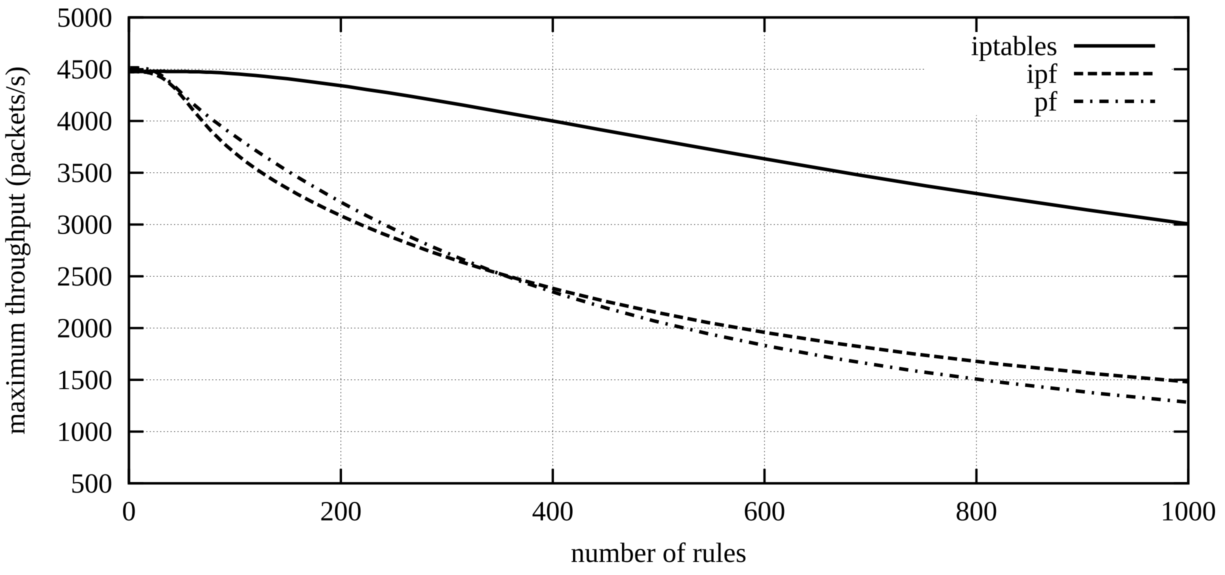

Figure 5: Maximum throughput with increasing number of rules

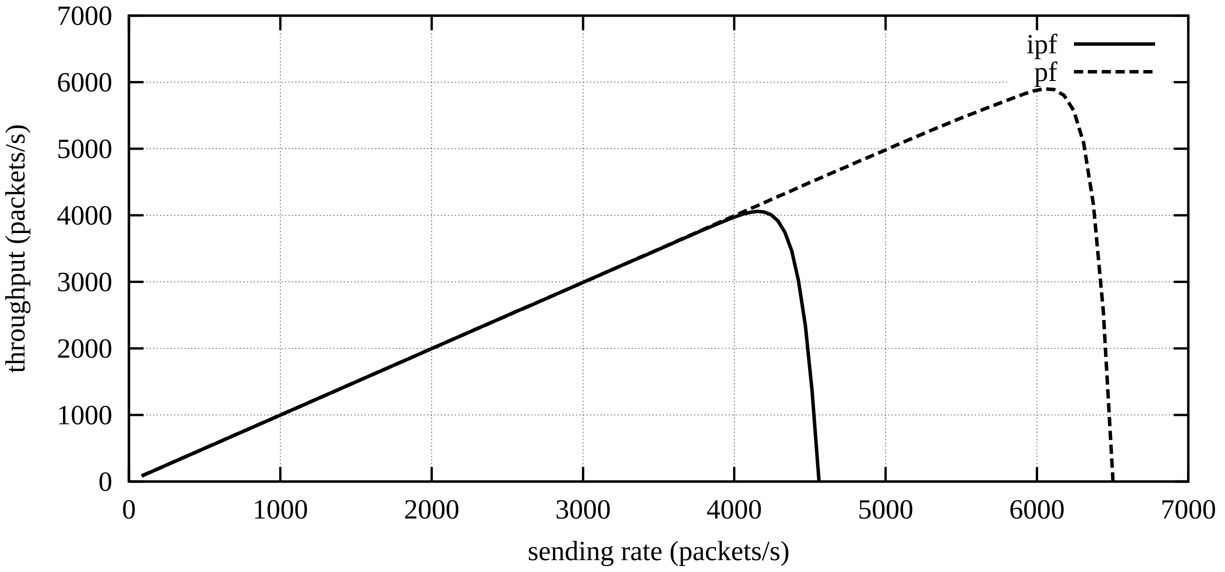

Figure 6: Stateful filtering with 20000 state entries (throughput)

The latency comparison for this test is shown in Figure 7. The latency increases as expected when the packet filters reach overload.

Similarly to the second test, the procedure is repeated for various state table sizes between one and twenty five thousand. Figure 8 shows the maximum lossless throughput rate as a function of the state table size. We notice that *pf* performs significantly better than *IPFilter* when the size of the state table is small. At ten thousand state table entries, *IPFilter* starts to perform better than *pf*, but the difference is not significant.

We expected Figure 8 to show the $O(1)$ behavior of hash table lookups for *IPFilter* and $O(\log n)$ of tree lookups for *pf*. However, it seems that the constant cost of hash key calculation is relatively high,

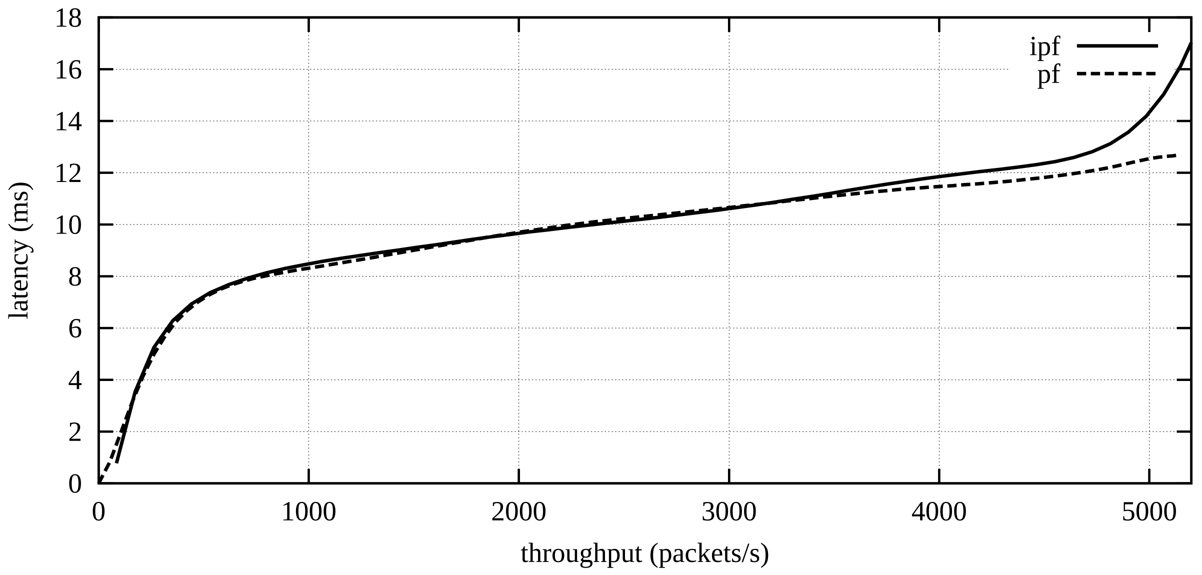

Figure 7: Stateful filtering with 20000 state entries (latency)

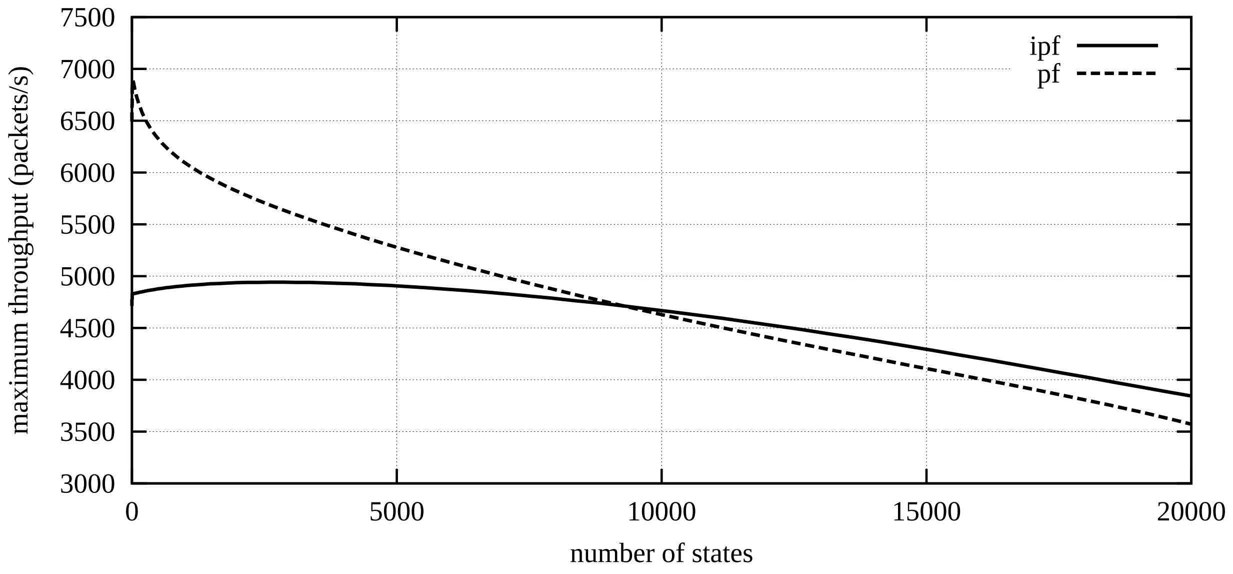

Figure 8: Maximum throughput with increasing number of states

and *IPFilter* still depends on n for some unknown reason.

Iptables has not been included in this benchmark because it does not do stateful filtering comparable to *pf* and *IPFilter*. The version of *iptables* that we tested employs *connection tracking* without any sequence number analysis for packets outside of the initial TCP handshake. While this is unsurprisingly faster, it would be an unfair performance comparison. There is a patch for *iptables* that adds sequence number checking, but it is still beta and is not included in the GNU/Linux distribution used for testing.

3.4 Discussion

The stateless benchmark indicates that rule set evaluation is very expensive in comparison to state table lookups. During the initial tests, *pf* was considerably slower in evaluating rules than *IPFilter*.

The slower performance is explained by the fact that *pf* used to evaluate the rule set three times for every packet that passes an interface: twice to look for *scrub* rules which determine whether IP and TCP normalization should be performed and once to look for *pass* and *block* rules.

This problem can be rectified by a simple optimization. It is sufficient to add two additional *skip-steps* for rule types *scrub* versus *pass/block* and the direction *in* versus *out*. This change which is now part of OpenBSD 3.1 improves the performance of *pf*'s rule set evaluation considerably, as shown in Figure 9.

The benchmarks measure only how packet filter performance scales for extreme cases to show the behavior of rule set evaluation and state table lookup algorithms, e.g., completely stateless and when all packets match state. In real-life, a firewall will perform different mixtures of these operations, as well as other operations that have not been tested, like creation and removal of state table entries and logging of blocked packets.

Also, real-life packets rarely require a complete evaluation of the rule set. All tested packet filters have mechanisms to reduce the number of rules that need to be evaluated on average. *Iptables* allows the definition of and jumps to *chains* of rules. As a result, the rule set becomes a tree instead of a linked list. *IPFilter* permits the definition of rule *groups*, which are only evaluated when a packet matches a *head* rule. *pf* uses *skip-steps* to automatically skip rules that cannot apply to a specific packet. In summary, *iptables* perform the best for stateless rules and *pf* performs the best when using stateful filtering.

4 Future work

There are still several areas in which *pf* may be improved. Plans for future work include among other things the following:

- application level proxies for additional protocols,
- authentication by modifying filter rules to allow network access based on user authentication,
- load-balancing, e.g., redirections translating destination addresses to a pool of hosts to distribute load among multiple servers,
- fail-over redundancy in which one firewall replicates state information to a stand-by firewall that can take over in case the primary firewall fails and
- further TCP normalization, e.g., resolving overlapping TCP segments, as described in the traffic normalization paper by Handley *et al.* [5].

5 Conclusions

This paper presented the design and implementation of a new Stateful Packet Filter and compared its performance with existing filters. The key contributions are an efficient and scalable implementation of the filter rule set and state table, automatic rule set optimization and unique features such as normalization and sequence number modulation.

The benchmarks show that the lower cost of state table lookups compared to the high cost of rule set evaluations justify creating state for performance reasons. Stateful filtering not only improves the quality of the filter decisions, it effectively improves filtering performance.

The new Stateful Packet Filter is included in OpenBSD 3.0 released in November 2001. The source code is BSD licensed and available in the OpenBSD source tree repository [7].

6 Acknowledgements

The OpenBSD packet filter is a collaborative work, and many developers have contributed code, ideas and support. In particular, I'd like to thank Artur Grabowski, Bob Beck, Dug Song, Jason Ish, Jason Wright, Jun-ichiro itojun Hagino, Kenneth R Westerback, Kjell Wooding, Markus Friedl, Mike

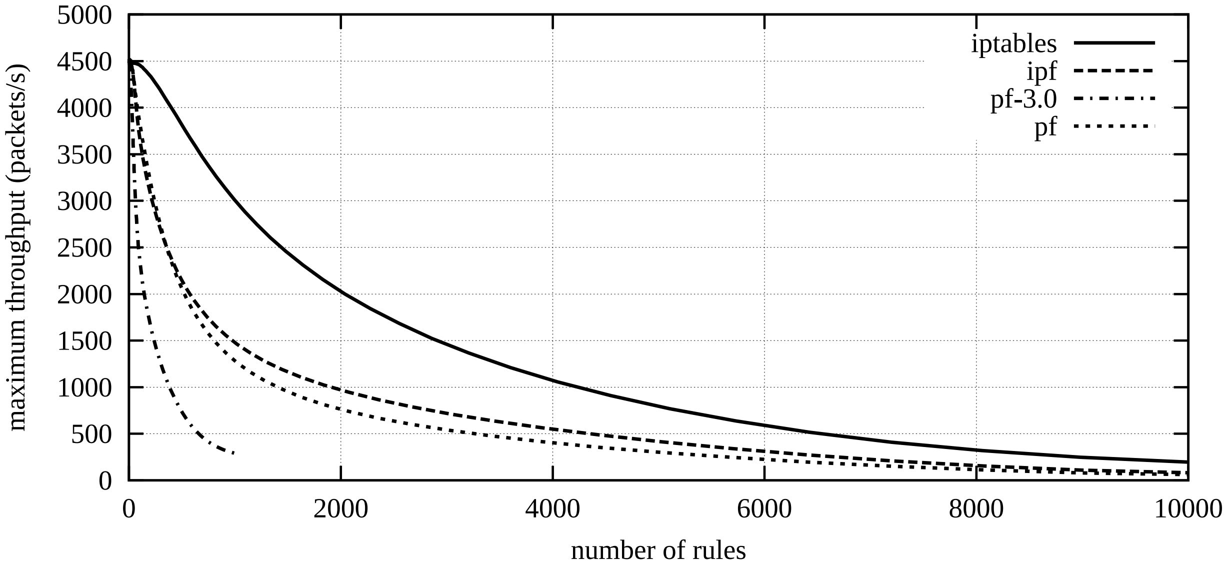

Figure 9: Maximum throughput with increasing number of rules

Frantzen, Niels Provos, Ryan McBride, Theo de Raadt, Todd C. Miller and Wim Vandeputte. Most of the equipment and time used to write this paper was funded by Systor AG.

References

[1] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. Internet RFC 2544, March 1999.

[2] David Watson Farnam Jahanian, G. Robert Malan and Paul Howell. Transport and application protocol scrubbing. In *Proc. Infocomm 2000*, 2000.

[3] Patricia Gilfeather and Todd Underwood. Fragmentation made friendly. In *http://www.cs.unm.edu/∼maccabe/SSL/frag/FragPaper1/Fragmentation.html.*

[4] C. A. Kent and J. C. Mogul. Fragmentation considered harmful. In *WRL Technical Report 87/3*, December 1987.

[5] C. Kreibich M. Handley and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium 2001*, 2001.

[6] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proc. 1993 Winter USENIX Conference, San Diego, CA*, January 1993.

[7] The OpenBSD project. http://www.openbsd.org/.

[8] Guido van Rooij. Real stateful tcp packet filtering in ip filter. In *http://www.madison-gurkha.com/publications/tcp_filtering/tcp_filtering.ps*, 2000.