Netfilter Performance Testing

József Kadlecsik
KFKI RMKI
kadlec@sunserv.kfki.hu

György Pásztor
SZTE EK
pasztor@linux.gyakg.u-szeged.hu

Abstract

This paper documents the results of the performance testing of netfilter, the firewalling subsystem of the Linux kernel. We compared the performance of two different hardware configurations and measured the throughput at plain routing, connection tracking, filtering and NAT. We also examined the dependency of the performance on the number or rules for iptables, nf-hipac and ipset.

1. Introduction

Linux is often used for firewalling and there are Linux distributions with the sole purpose of building a network firewall based on netfilter[1], which provides the firewall functionalities of the Linux kernel 2.4 and above. However it is very hard to find published performance results of this critical segment of the Linux kernel. Our goal was to fill this gap and measure the performance of Linux and netfilter at every major stage of packet filtering: plain routing, connection tracking, filtering and NAT. Two different hardware configurations were compared and performance dependency on the number of rules was examined using iptables, nf-hipac[2] and ipset[3] as well.
In Section 2 we describe the chosen methodology of the testing. Section 3 details the traffic generator environment, the tested hardware and the used software. In Section 4 we present the data we measured and an analysis of the results. Finally, Section 5 sums up our experiences during the presented performance testing.

2. Firewall performance measurement

Modern packet filtering firewalls are stateful, that is they can associate the packets to the connection or stream the packets belong to. Such a feature makes possible to build more natural and secure firewall rules compared to that of the stateless firewalls. In order to support the functionality, stateful firewalls must maintain a state table of the connections flowing through the firewall so that the packets can be associated with the entries in the table. The most expensive operation for such systems is finding out that a packet does not belong to any already existing connection and thus a new entry must be allocated, initiated and added to the state table. The operations on the new entries will impose an upper limit on the rate at which new connections can successfully be opened up through stateful firewalls.

The dominant protocol of today Internet traffic is TCP. Even when it were much more simpler and easier to generate UDP traffic, more relevant results can be gained by generating TCP streams during the testing. Also, due to the complexity of the protocol, tracking the TCP

connections needs more resource and time from the firewalls. As our goal is to measure the performance of a stateful firewall we believe plain packet and byte counters are not completely enough for comparison and the number of connections, more specifically the number of the **new** connections per second which can be handled by the firewall is a better performance indicator. Therefore our whole test environment must be tailored towards to generate high number of parallel TCP connections and during the tests we must generate controlled number of new TCP connections per second, counting the successfully completed sessions. However when generating TCP traffic, there are some constrains on the client machines[4], which must be taken into account and dealt with:

- Size of available TCP port range:
  When connecting to the same server on the same port, there are 64,512 non-privileged ports available on the client side as source ports. According to RFC793[5], a port cannot be reused until the TCP_TIME_WAIT state expires. The recommended timeout value in the RFC is 4 minutes, which would mean 268 new request per second at the maximum. In the Linux kernel the timeout value of the TCP_TIME_WAIT state is around 1 minute, which means a maximum of 1075 new request per second.

- Number of open file descriptors per process:
  Linux imposes a default limit of maximum 1024 open file descriptors for processes. If the full TCP session timeout is five seconds, then at the worst case maximum 204 new request per second can be achieved. By lowering the timeout value we discard slow connections, so we have to find a way to increase the limit of open file descriptors for the client software.

- Socket buffer memory:
  Each TCP connection requires read and write socket buffer memory from the kernel. For example assuming 16kB receive buffer per client and an available total receive buffer space of 40MB, when all the buffers are full, the client hits the limit of 2560 concurrent connection.

Therefore in order to avoid the possible bottlenecks, we tuned kernel and interface parameters on the client and the server machines. In Appendix B we list the shell script and the parameters we applied during the tests.

3. The environment, hardware and software

As traffic generator environment, we used the public machines of the Library of the University of Szeged, outside of the opening hours. After the library closed, the machines were triggered to reboot and load a Linux kernel and system image into memory, and before the opening the library, the machines were rebooted again for the normal daily operation. Half of the total number of 160 machines were configured as clients, half as servers during the tests.
Figure 1 below shows the physical topology of the network: clients are the machines with IP address from the network 192.168.0.0/24, while server addresses were assigned from 192.168.1.0/24. All the clients were aggregated by the switch named SW-RS, while the servers by SW-RSZ into different VLAN-s. We specified the path costs on the switch interfaces explicitly, so clients and servers were explicitly separated by breaking the spanning tree at the 'x' signs as shown on the figure. The duplicated optical gigabit connection between SW-RS and SW-RSZ was unified into an etherchannel link, therefore the theoretically

maximal throughput between the main switches was 2 Gbit/s. The test firewalls were connected directly via copper Gbit to switch SW-RS.

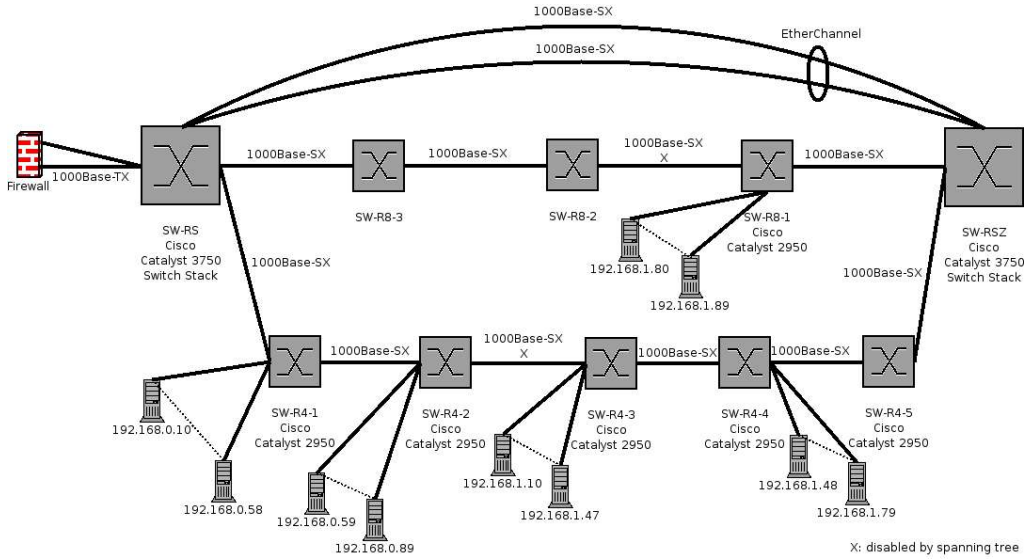

Figure 1 Physical topology

The logical topology is much more straightforward and shown on Figure 2:

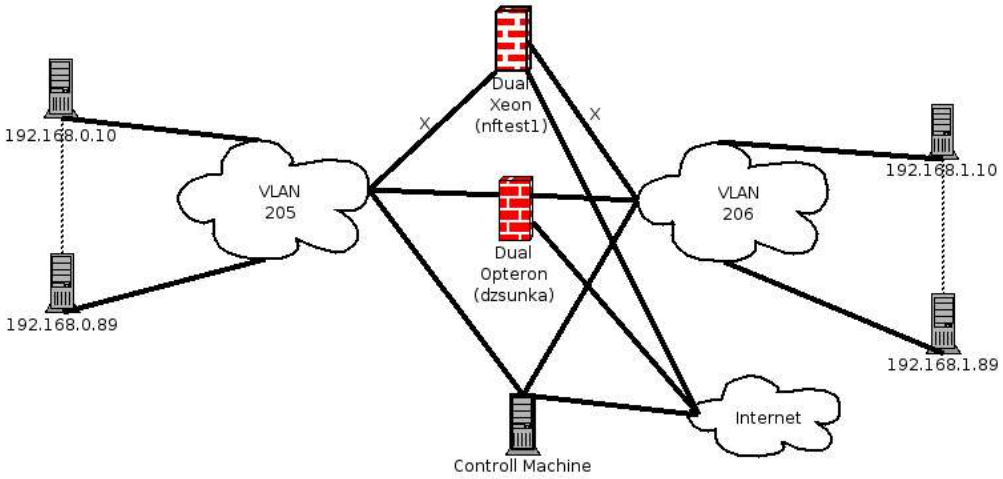

Figure 2 Logical topology

The client and server machines have got the the same hardware configuration: 2.9 GHz Intel Pentium IV CPU, 512 MB RAM and broadcom (tg3) GigabitEthernet interface integrated onto the motherboard.

The client machines ran httperf[4] as traffic generator software and the servers used boa[6] as http server to respond. At client side httperf was recompiled with 65536 maximal open file descriptors[7], while logging, CGI, mime support were disabled in boa in order to avoid any unnecessary overhead at server side. Each client connected to its server pair and downloaded the same file, which fit into one packet (one complete TCP session for such a HTTP download requires about ten packets).

The tested firewall hardware configurations were the following:

- nftest1, codename “Xeon”:
  - Dual Intel Xeon 2.4GHz
  - 2GB DDR RAM, 200MHz
  - ServerWorks GC-LE chipset
  - Intel 82545EM Gbit Ethernet, 64bit, 66MHz

- dzsunka, codename “Opteron”
  - Dual AMD Opteron 2.2GHz
  - 4GB DDR RAM, 333MHz
  - AMD-8131 chipset
  - Intel 82546EB Gbit Ethernet, 64bit, 66MHz (dual)

Please note, while we refer the machines as Xeon and Opteron for the sake of simplicity below, the processor type is just one factor in the performance differences: the memory speed, the internal bus and the improved ethernet interface in Opteron are significant factors as well.

On nftest1 we installed both 2.4.29 and 2.6.11 while on dzsunka we ran 2.6.11 alone (most recent kernels at the time of the testing). In all cases the kernels were stripped down to the bare minimum to support the firewalling functionality alone and we used modularized kernels.

There was a management machine (see Figure 2), from which we could transfer files to/from all the machines, could inititate the test series and collect the log files.

The test series were performed by a handful of shell scripts on the management, server and client machines, and were initiated by the following shell script skeleton:

```
START=5000      # Initial new connection rate
STEP=5000       # at every step the rate increased by this
END=50000       # last tested rate
TIMEOUT=5       # httperf timeout parameter
CLIENTS=80      # number of client machines

nf_test() {
     DIR=$HOME/nf-test/results/$1
     mkdir -p $DIR
     conns=$START
     while [ $conns -le $END ]; do
          for repeat in 0 1 2; do
               initialize log collectors on firewall
               start httperf on clients in background
               sleep five minutes
               collect logs from firewall to $DIR
               collect httperf results from clients to $DIR
          done
          conns=$((conns+$STEP))
     done
}
```

```
for kernel in 2.6.11 ...; do
     reboot firewall
     wait for firewall to come up
     nf_test $kernel
done
```

All the tests were repeated three times and on the graphs we display the average calculated from the repeated tests. One test took three minutes and we waited two minutes between two tests to let late packets leave the system and connections time out. From the client machines the httperf outputs were collected. From the tested firewall we collected the following log results:

- kernel.log
- content of */proc/net/dev* and */proc/interrupts* before and after the test
- content of */proc/sys/net/ipv4/netfilter/ip_conntrack_count* sampled at every five seconds
- log generated by *atsar* during the test

The reply rate graphs below were generated from the httperf result files where we summed up the average replay rates recorded by the individual clients. The packet per second graphs were created from the transmitted packet counter entries from the samples of */proc/net/dev* saved before and after every the test on the firewall.

4. Test results

4.1 Routing performance

The first test series covered the measurement of raw throughput when no firewall functionality was enabled and the firewalls worked as routers alone. Thus the measurement of plain routing performance can serve as a reference performance for the firewall testing.

Figure 3 and 4 show the results for Xeon running 2.4.29 and 2.6.11 (both case NAPI enabled) and Opteron with 2.6.11, first with NAPI disabled, then enabled.

The breaking point for Xeon is at about 30,000 new requests per second, while according to Figure 3, we could not reach the breaking point for Opteron. On Xeon, 2.6.11 performs better under high load than 2.4.29 and from the lines for Opteron we can see that NAPI can help to improve performance on 2.6.11.

In order to reach the limits of Opteron, we ran another test with request rates up to 80,000 new request/s, which is about the upper limit of our testing environment. Also, we examined the impact of the different values of the RxDescriptor and TxDescriptor parameters of the e1000 driver. We tested the machine with the default Rx|TxDescriptor value of 8 and with the values 512 and 4096 as well, where the latter is the maximum value supported by the hardware.

Figure 5 and 6 show the results of these tests. We can see that the breaking point of Opteron with NAPI enabled is around 55,000 new connections/s handled completely and the maximal packet rate is about 700,000 pps. Increasing the Rx|TxDescriptor parameters of the interface to 512 could help to improve the performance slightly, but using the maximal values resulted a noticeable drop in the performance.

4.2 Conntrack, filtering and NAT

At the firewall tests we aimed to investigate the effect of the different layers on the overall performance. Therefore first we examined plain connection tracking without any filtering or NAT rule applied. Then the second test series comprised connection tracking and filtering with the following rules:

```
iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -d 100.100.100.1 -j DROP
...
iptables -A FORWARD -d 100.100.100.10 -j DROP
iptables -A FORWARD -p tcp --dport 80 -m state --state NEW -j ACCEPT
```

i.e. standard stateful rules with ten non-matching rules before the one which allows the new connections to build up. Finally in the NAT tests the connection tracking and filtering above was extended by the NAT rule:

```
iptables -t nat -A POSTROUTING -s 192.168.0.0/24 \
                -p tcp --dport 80 -j NETMAP --to 10.10.10.0/24
```

which forced the firewall to NAT all client requests.

Figure 7, 8 displays the results on conntrack: the maximal performance halved compared to the plain routing case and the maximal new connection rate is around 25,000 new connections/s, while the packet rate is about 330-340,000 pps. It is clear that connection tracking is an expensive operation, which requires a lot of resources from the system.

Figure 9 and 10 shows connection tracking and filtering together and the graphs do not show a significant drop in performance compared to the conntrack-only case. Apparently, non-excessive filtering does not pose a real performance challenge for netfilter/iptables.

On Figure 11, 12 we can see the full-blown setup: connection tracking, filtering and NAT together. As it was assumed, NAT means a burden over the system: the request rate lowered to 20,000 new request/s or below and the packet rate is slightly above 200,000 pps. In order to get a better view on the impact of filtering and NAT over connection tracking, Figure 13 show the results for Opteron from the previous graphs.

Finally, Figure 14 shows the number of concurrent connections handled by conntrack when Opteron handled 30,000 new request/s during the lifetime of one single test. As the graph displays, the system handled almost 3,500,000 concurrent connections at the peak.

4.3 Performance dependency on the number of rules

It is well known that netfilter/iptables does not scale well if one wants to use large number of rules in a single chain. The reason of the problem lies in the fact that the rules are processed in netfilter/iptables one after another, linearly.

Several solutions emerged over the time:

- nf-hipac: http://www.hipac.org/

  nf-hipac uses binary tree structures to store and evaluate the rules. The user interface closely follows the syntax of iptables and besides the native nf-hipac matches, it can use non-native matches from iptables. It supports the filter table alone and there seems to be

no port to the 2.6 kernel series. Unfortunately there is no documentation available on the algorithms behind nf-hipac.

- Compact Filter (CF)[8]: http://www.cs.aau.dk/~mixxel/cf

  CF implements a so called interval decision diagram based firewall over the framework of netfilter. In CF, the firewall rules are optimized in userspace and the compact ruleset is transferred to the kernel. It works with 2.6 kernels but a minimal set of matching is supported only: protocol, source/destination addresses and ports.

- iptables with classifiers[9]: http://www.geocities.com/hamidreza_jm/

  An interesting approach, which adds support to different kind of classifiers to iptables. Besides the original linear classifier, it implements a so called tuple classifier with hashing behind it. Patches are against the 2.6 kernel tree.

- ipset: http://ipset.netfilter.org/

  ipset is not a complete replacement or extension of iptables itself, but a new matching fully integrated into iptables. ipset supports matching of IP addresses, netblocks or port numbers stored in bitmaps, hashes or trees. Both the 2.4 and 2.6 kernel trees are supported.

In the present testing we compared iptables, nf-hipac and ipset running on Xeon with Linux 2.4.29. The systems were tested using the equivalent of the following pseudo-rules:

```
-m state --state ESTABLISHED,RELATED -j ACCEPT

 < N non-matching IP address rule >

-p tcp --dport 80 -m state --state NEW -j ACCEPT
```

where N started at 16 and doubled until we reached 16384 non-matching rules. (In order to test the worst case, iphash type of set was used in the ipset tests.)

Figure 15 and 16 display the reply rate and pps which could be achieved. The lines for iptables on both figures show clearly the non-scaling behaviour of iptables. However both nf-hipac and ipset performed almost indifferently with regard of the number of rules. ipset was a tiny bit better than nf-hipac, but ipset is more lighter and simpler than nf-hipac. The last figure shows the required time to add the given number of rules to the kernel. Again, iptables suffers from its linear algorithms (which produces exponential behaviour in rule-addition due to the cumulating effect) while nf-hipac and ipset are immune from such problems.

5. Summary

In this paper we presented the results of a performance testing of Linux netfilter. We tested kernel releases from both the 2.4 and 2.6 trees and compared the performance of two different hardware configurations. We examined raw routing performance and the throughput when connection tracking, connection tracking with filtering and connection tracking with filtering and NAT were enabled. We also tested how iptables, nf-hipac and ipset behave as one increase the number of filtering rules.

The results show that netfilter/iptables is a viable solution for firewalling, even in gigabit environment. There is space for improvements in the connection tracking and NAT subsystems and with proper chain structures or using the alternative solutions one can avoid the bottlenecks of iptables at handling large list of rules.

Acknowledgements

We would like to express our gratitude to the Library of University of Szeged for letting us to use their machines and environment.

References

[1] Netfilter: firewalling, NAT and packet mangling for Linux, http://www.netfilter.org

[2] David Mosberger, Tai Jin: httperf – A Tool for Measuring Web Server Performance

[3] Boa Webserver: http://www.boa.org

[4] Patrick O'Rourke, Mike Keefe: Performance Evaluation of Linux Virtual Server

[5] RFC 793: Transmission Control Protocol

[6] nf-hipac: High Performance Packet Classification: http://www.hipac.org/

[7] Compact Filter: An IDD Based Packet Filter for Linux: http://www.cs.aau.dk/~mixxel/cf/

[8] New iptables with classifiers: http://www.geocities.com/hamidreza_jm/

[9] IP sets: http://ipset.netfilter.org/

Appendix A

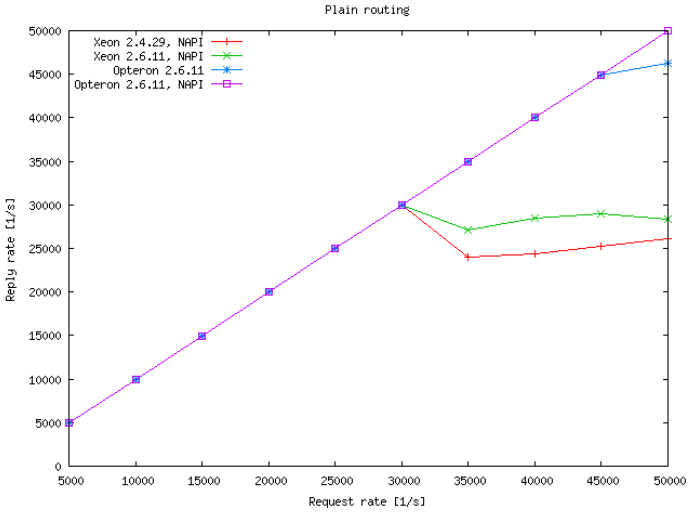

Figure 3 Routing

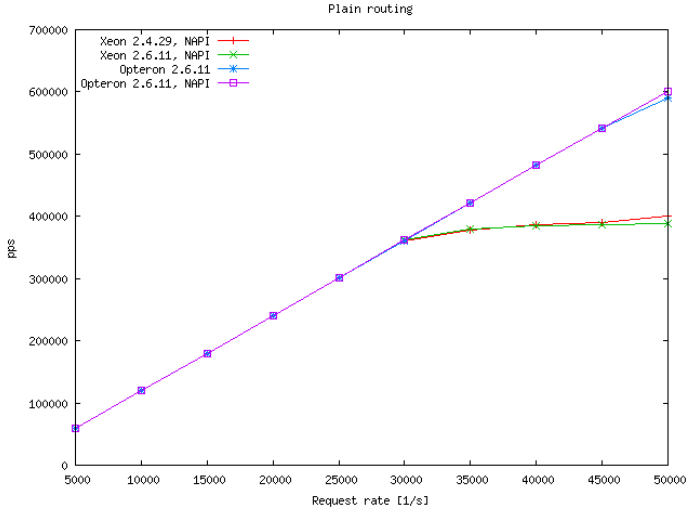

Figure 4 Routing

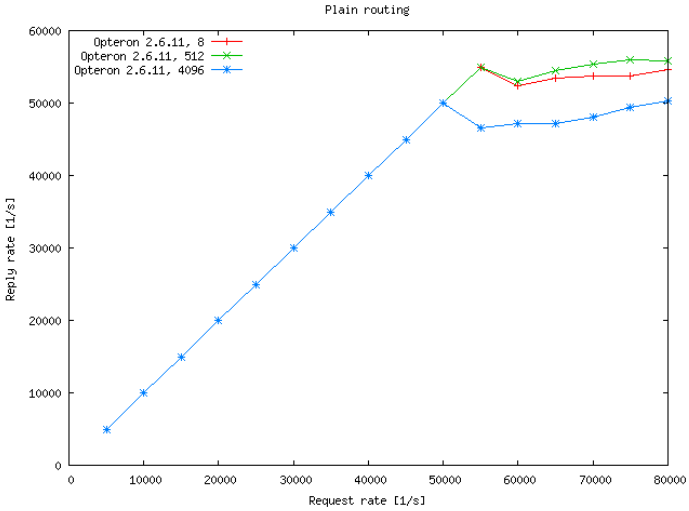

Figure 5 Routing, Opteron

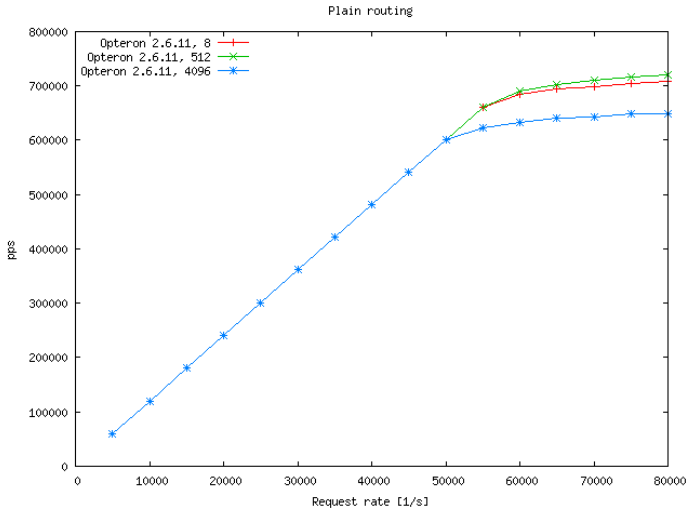

Figure 6 Routing, Opteron

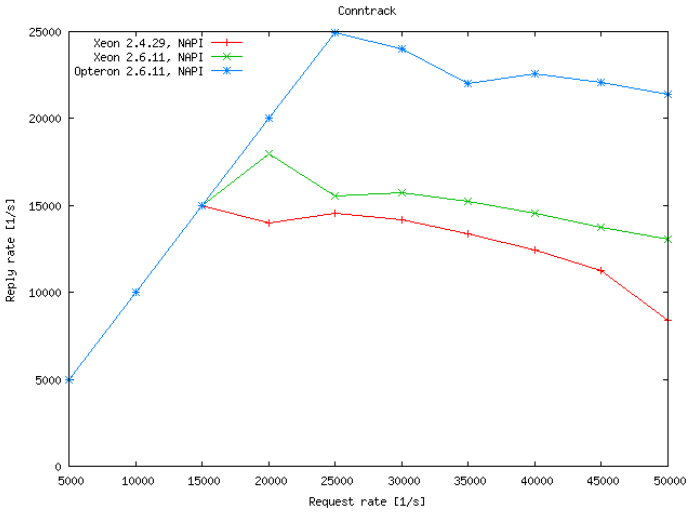

Figure 7 Conntrack

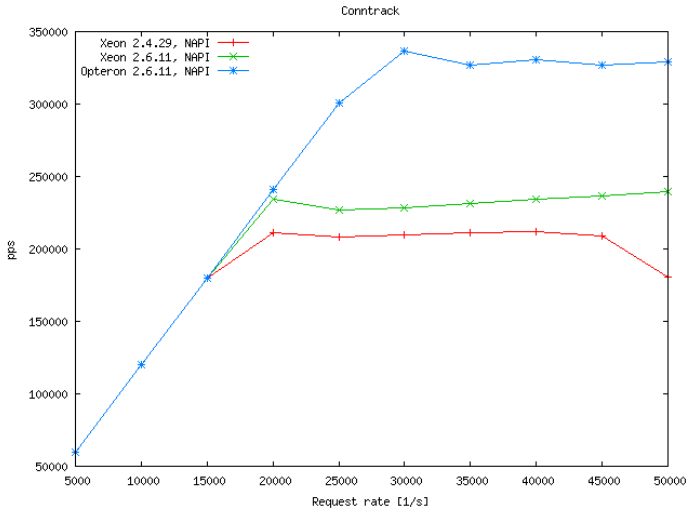

Figure 8 Conntrack

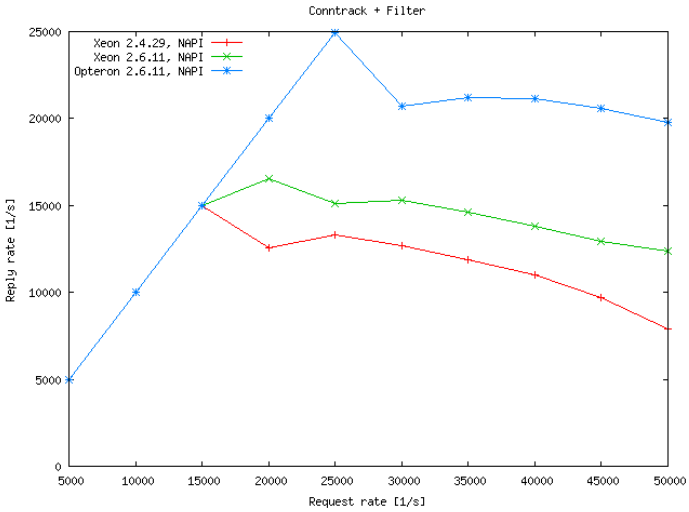

Figure 9 Conntrack + filter

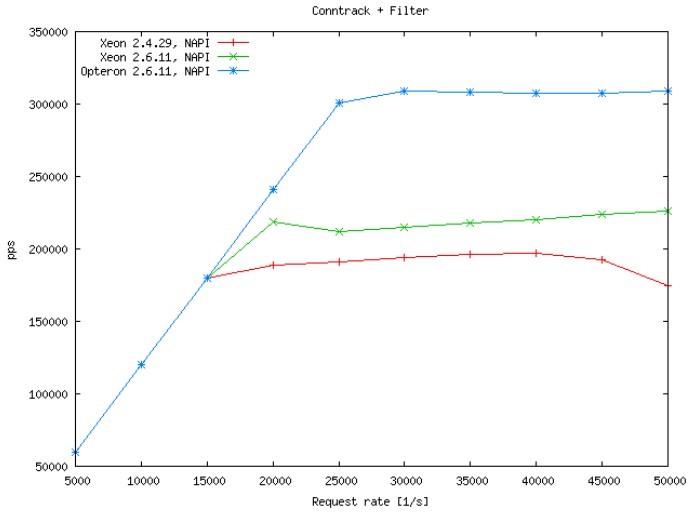

Figure 10 Conntrack + filter

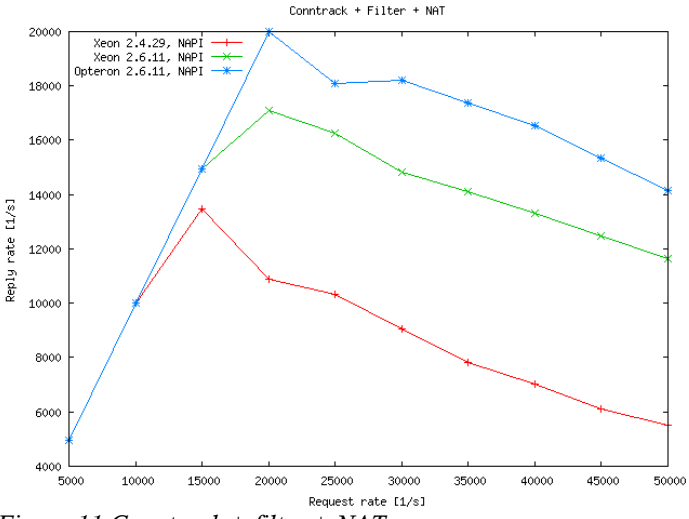

Figure 11 Conntrack + filter + NAT

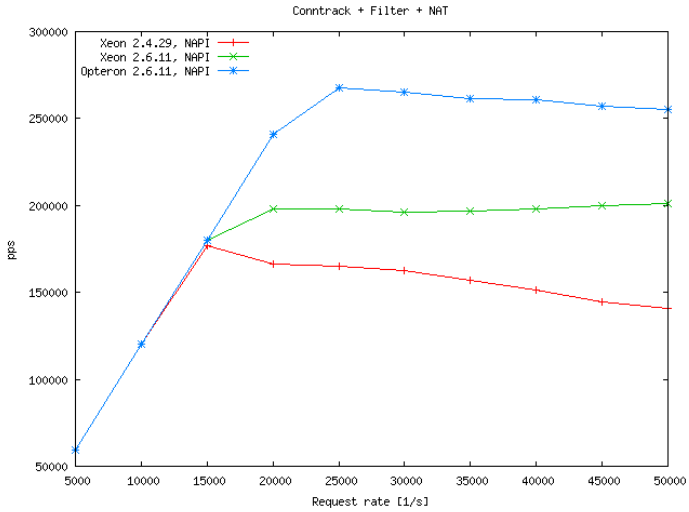

Figure 12 Conntrack + filter + NAT

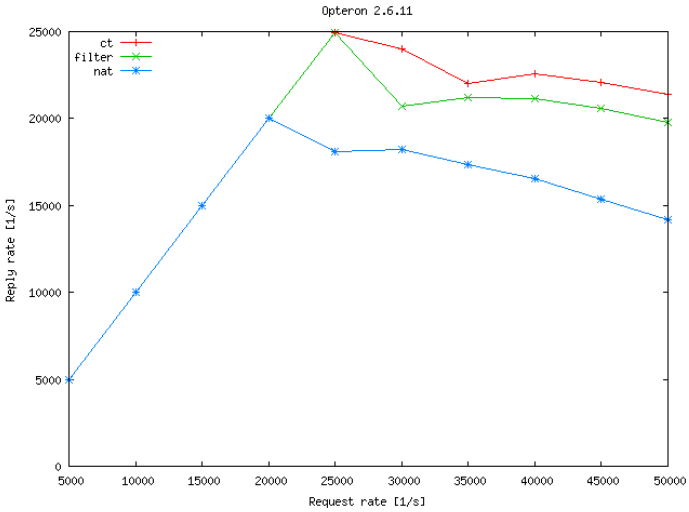

Figure 13 Conntrack/filter/NAT comparison

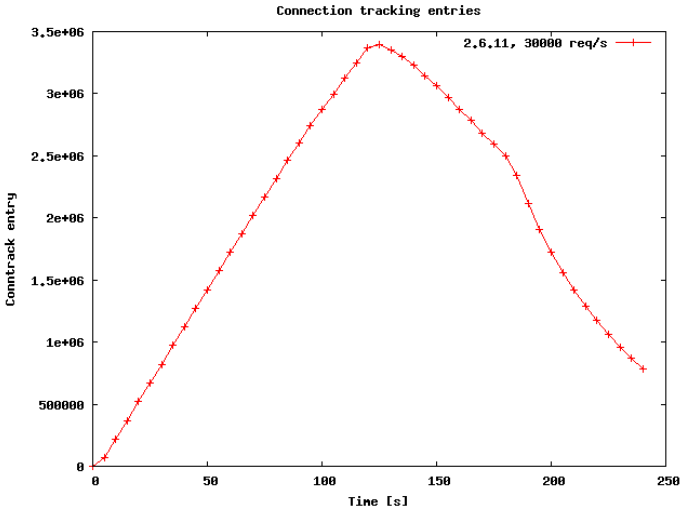

Figure 14 Number of concurrent connections

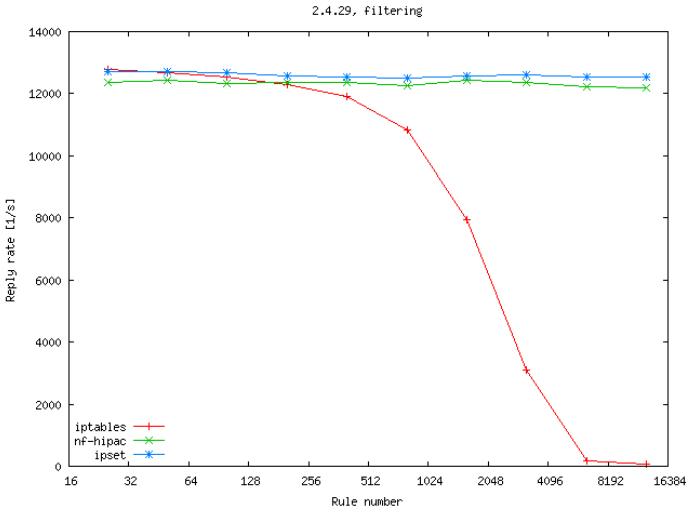

Figure 15 iptables, nf-hipac, ipset

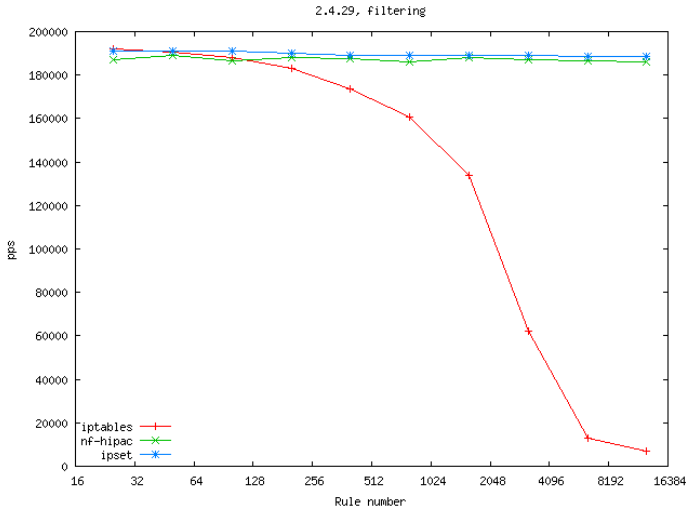

Figure 16 iptables, nf-hipac, ipset

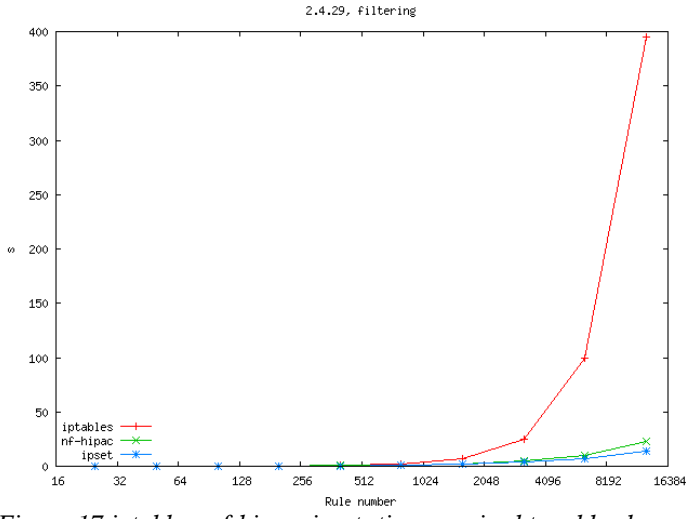

Figure 17 iptables, nf-hipac, ipset: time required to add rules

Appendix B

Script to tune kernel and interface parameters on the test machines:

```
#!/bin/sh

# 4KB send buffer, 20,480 connections max at worst case
echo 83886080  > /proc/sys/net/core/wmem_max
echo 83886080  > /proc/sys/net/core/wmem_default

# 16KB receive buffer, 20,480 connections max at worst case
echo 335544320 > /proc/sys/net/core/rmem_max
echo 335544320 > /proc/sys/net/core/rmem_default

# Max open files
echo 65536 > /proc/sys/fs/file-max

# Fast port recycling (TIME_WAIT)
echo 1 >/proc/sys/net/ipv4/tcp_tw_recycle
echo 1 >/proc/sys/net/ipv4/tcp_tw_reuse

# TIME_WAIT buckets increased
echo 65536 > /proc/sys/net/ipv4/tcp_max_tw_buckets

# FIN timeout decreased
echo 15 > /proc/sys/net/ipv4/tcp_fin_timeout

# SYN backlog increased
echo 65536 > /proc/sys/net/ipv4/tcp_max_syn_backlog

# SYN cookies enabled
echo 1 > /proc/sys/net/ipv4/tcp_syncookies

# Local port range maximized
echo "1024 65535" > /proc/sys/net/ipv4/ip_local_port_range

# Netdev backlog increased
echo 100000 > /proc/sys/net/core/netdev_max_backlog

# Interface transmit queuelen increased
ifconfig eth0 txqueuelen 10000
```