Optimizing the Performance of the Iptables Stateful NAT44 Solution

Gábor Lencse, Keiichi Shima

Abstract—**The stateful NAT44 performance of iptables is an important issue when it is used as a stateful NAT44 gateway of a CGN (Carrier-Grade NAT) system. The performance measurements of iptables published in research papers do not comply with the requirements of RFC 2544 and RFC 4814 and the usability of their results has serious limitations. Our Internet Draft has proposed a benchmarking methodology for stateful NATxy (x, y are in {4, 6}) gateways and made it possible to perform the classic RFC 2544 measurement procedures like throughput, latency, frame loss rate, etc. with stateful NATxy gateways using RFC 4814 pseudorandom port numbers. It has also defined new performance metrics specific to stateful testing to quantify the connection setup and connection tear down performance of stateful NATxy gateways. In our current paper, we examine how the performance of iptables depends on various settings, and also if certain tradeoffs exist. We measure the maximum connection establishment rate, throughput and tear down rate of iptables as well as its memory consumption as a function of hash table size always using 40 million connections. We disclose all measurement details and results. We recommend new settings that enable network operators to achieve significantly higher performance than using the traditional ones.**

Index Terms—**benchmarking, iptables, netfilter, optimization, performance, stateful NAT44.**

I. INTRODUCTION

THE depletion of the public IPv4 address pool of IANA in 2011 has presented the ISPs (Internet Service Providers) with a dilemma: either they deploy IPv6 as soon as possible or they use CGN (Carrier-Grade NAT). We believe that the first one is the only workable solution in the long run, but we also experience that the transition to IPv6 is happening rather slowly for various reasons [1], and we estimate that IPv4 will be with us for decades. Therefore, stateful NAT44 (also called NAPT: Network Address and Port Translation) gateways will also be needed for a long time. The *Netfilter Framework* [2] of the Linux kernel (usually called *iptables* after the name of its command line management tool) is a widely used solution for this purpose.

Submitted: December 14, 2022.
This work was supported by the International Exchange Program of the National Institute of Information and Communications (NICT), Japan.
Gábor Lencse is with the Department of Telecommunications, Széchenyi István University, Győr, Hungary (e-mail: lencse@sze.hu).
Keiichi Shima is with the SoftBank Corporation, Minato-ku, Tokyo, Japan.

We are aware that in some areas of application, iptables is gradually replaced by *nftables*. The latter has advantages, when a high number of rules are used and they are often reconfigured, but it did not became an industry standard yet [3]. When implementing CGN, there is no need for a high number of rules and they are very rarely reconfigured, thus iptables is still appropriate for this purpose. What really matters for the ISPs, it is the performance of the stateful NAT44 translation. To that end, iptables is a good choice: the iptables stateful NAT44 solution outperformed the Jool NAT64 solution by an order of magnitude in throughput and its performance also scaled up much better with the number of active CPU cores and showed much less degradation with the number of connections than Jool according to our measurements [4]. However, we have also experienced that the performance of iptables highly depends on certain parameters.

The aim of our current paper is to investigate how the performance of iptables depends on various settings, and also to examine what kind of tradeoffs exist, and thus recommend optimal settings depending on the actual performance needs and hardware parameters of the ISPs.

The remainder of this paper is organized as follows. In Section II, we make a survey how iptables is used in the current research papers and how its performance is analyzed and/or optimized. In Section III, we give a short summary of the state of the art methods for measuring the performance of stateful NAT44 gateways. In Section IV, we overview some relevant details of iptables including its tunable parameters and their recommended values as well as how they influence the memory consumption of iptables. In Section V, we disclose our measurements and their results. In Section VI, we discuss our results and give our recommendations to optimize the performance of iptables. Section VII is an additional case study in which we examine the performance of nftables. Section VIII concludes our paper.

II. RELATED WORK

A. Peer-reviewed Papers

We have surveyed, how iptables appears in research papers from the latest years. We found that it is usually mentioned as a firewall and not as a stateful NAT44 solution. And the methods used for measuring its performance does not comply with the relevant IETF RFCs, please see their requirements in Section III.A.

DOI: 10.36244/ICJ.2023.1.6

```
              +------------+
              |            |
+-------------|   Tester   |<----------+
|             |            |           |
|             +------------+           |
|                                      |
|             +------------+           |
|             |            |           |
+------------>|    DUT     |-----------+
              |            |
              +------------+
```

Fig. 1. Test setup for benchmarking network interconnect devices. [9]

For example, Melkov et al [3] compared the performance of iptables and nftables using very high number of rules (up to several times 10,000). In contrast with the common view, they have found that iptables significantly outperformed nftables. Depending on the actually examined chain, the throughput of nftables significantly deteriorated around 5,000 or 10,000 rules, whereas iptables could sustain a good performance up to 20,000 or 40,000 rules. We note that they measured "TCP throughput" using iperf, and displayed the results in Mbps.

Gandotra and Sharma [5] also measured the firewall performance of iptables using 200, 500, 1000, 5,000, and 10,000 number of rules, TCP traffic with 1024 bytes packets size, multiple packet rates starting from 1,000pps increased by 1,000pps steps to 8,000pps, and test durations of 30s and 120s. As for measurement tool, they used D-ITG (Distributed Internet Traffic Generator).

Taga at al. [6] used iptables for testing their firewall traversal method. As for measurement method, they downloaded HTML files with different sizes and measured the download time.

B. Other Sources

In order to find more closely related sources to our topic, we have lowered the bar and did not require peer-reviewed papers.

Thus, we found a really closely related writing of Andree Toonk [7]. One of his tests was a stateful NAT44 performance measurement using a single iptables rule and 10,000 network flows. For the measurements, he used a DPDK-based packet generator, but the exact details of the measurements (how the bidirectional traffic was generated) are not disclosed. Using two 3.2GHz Intel Xeon Gold 5218 CPUs (in all 64 cores were available using hyper-threading) he managed to achieve a total of 5.9Mpps using bidirectional traffic. It does not turn out, if it was a lossless rate or not.

Whereas the above result is not bad, it definitely shows that there is room for performance optimization, as we achieved 5.3Mpps using only 16 cores of a 2.1GHz Intel Xeon E5-2683 v4 CPU even though we handled 1.56M connections (instead of only 10k) [4]. We note that our result is RFC 2544 [9] compliant throughput (non-drop rate). According to our measurements, the performance of iptables scaled up quite well with the number of CPU cores: when 4M connections were used and the number of active CPU cores was increased from 1 to 16, its *maximum connection establishment rate* (please refer to Section III.C) and *throughput* scaled up from 223.5kcps (connections per second) and 414.9kfps (frames per second) to 2,383kcps and 4,557kfps, respectively, thus the increase was more than tenfold [4]. We have also examined, how the performance of iptables degrades with the number of connections. In the range where we could increase the *hash table size* (please refer to Section IV.A) proportionally with the number of connections, the performance of iptables degraded only slightly with the 64-fold increase of the number of connection: when the number of connections were increased from 1.56M to 100M, its maximum connection establishment rate and throughput decreased from 2.406Mcps and 5.326Mfps to 2.237Mcps and 4.516Mfps, respectively. However, the degradation was more significant, when the built-in limitations of iptables prevented us from increasing the *hash table size* proportionally with the number of connections [4]. This is why we believe that it is worth examining how to optimize the parameters of iptables to provide ISPs with a high performance stateful NAT44 solution.

Theoretically, the reimplementation of iptables in eBPF could significantly outperform the native iptables. However, the measurement results of Massimo Tumolo show that it happens only if the number of the rules is above 100 [8]. It can happen, if iptables is used as a firewall. However, in our case, iptables is used as a stateful NAT44 gateway. Here the number of rules is very low (one or a few).

III. BENCHMARKING METHODOLOGY FOR STATEFUL NAT44 GATEWAYS

A. Benchmarking Methodology for Network Interconnect Devices

There is a long established benchmarking methodology for network interconnect devices defined by a series of IETF (Internet Engineering Task Force) RFCs. Commercial network performance tester vendors follow the requirements of RFC 2544 [9] for more than two decades. Its aim is to facilitate the measurement of the performance of network interconnect devices in an objective way. To that end it defines the most important conditions of the measurements to prevent gaming (or tricking or more openly: cheating), including:

- Test setup
- DUT (Device Under Test) settings (it may not be optimized for the given task)
- Test frame format and frame sizes (e.g. for Ethernet: 64, 128, 256, 512, 1024, 1280, 1518 bytes)
- Measurement procedures (throughput, latency, frame loss rate, back-to-back frames, system recovery, reset)
- Duration of the test (minimum 60s for throughput test)
- Requirement of testing with bidirectional traffic
- Usage of UDP as transport layer protocol
- Testing with a single IP address pair and also with 256 destination networks when routers are benchmarked.

As for test setup, the one shown in Fig. 1 should be used by default. Although the arrows are unidirectional, bidirectional traffic should be used as written above.

```
              +---------------------------------------+
     10.0.0.2 |Initiator                     Responder| 198.19.0.2
+-------------|                Tester                 |<------------+
| private IPv4|                          [state table]| public IPv4 |
|             +---------------------------------------+             |
|                                                                   |
|             +---------------------------------------+             |
|    10.0.0.1 |                 DUT:                  | 198.19.0.1  |
+------------>|        Stateful NAT44 gateway         |-------------+
  private IPv4|     [connection tracking table]       | public IPv4
              +---------------------------------------+
```

Fig. 2. Test setup for benchmarking stateful NAT44 gateways. [14]

From among the performance metrics, *throughput* is the most important one for us. It is defined as the highest constant frame rate, at which all frames can be forwarded by the DUT. Its measurement procedure requires that *test frames* are transmitted by the Tester through the DUT in both directions at least for 60 seconds at a constant frame rate, and the Tester counts the number of received test frames:

- If the number of the received test frames equals the number of the transmitted test frames then the frame rate is increased, and the test is rerun.
- If the number of the received test frames is less than the number of the transmitted test frames then the frame rate is decreased, and the test is rerun.

Whereas this wording facilitates various search algorithms, usually a binary search is executed using 0 and the maximum frame rate for the media as the starting interval.

Theoretically, it can be said that RFC 2544 is IP version independent, but in practice, it uses IPv4 addresses. The media types for which it defined the maximum frame rates in its appendix, also show its age.

As time passed by, the Benchmarking Working Group of IETF has produced further important RFCs. One of them is RFC 4814 [10]. It requires the usage of pseudorandom port numbers with uniform distribution over the following ranges:

- Source port number range: 1,024–65,535
- Destination port number range: 1–49,151

Without that requirement, the very same test frames could be sent, which was very convenient for the tester vendors, but it did not reflect the nature of the Internet traffic. Moreover, pseudorandom port numbers are necessary to support RSS (Receive-Side Scaling, also called multi-queue receiving) [11], because if the same source and destination IP addresses are used for each packet, then only the port numbers can ensure entropy for the hash function to distribute the interrupts of packet arrivals among the several cores of a contemporary CPU.

We note that there were two further important RFCs published. RFC 5180 [12] is mainly an IPv6 update regarding IPv6 specificities, but it also contains maximum frame rates for some media types being contemporary at the time of its writing. However, it excludes IPv6 transition technologies from its scope. They are covered by RFC 8219 [13]. It has kept the requirement of testing with bidirectional traffic, but it also introduces testing with single directional traffic as an optional measurement. We believe that the asymmetry of the amount of Internet traffic in download and upload directions is a good rational for testing also with unidirectional traffic, and it is worth extending it to the benchmarking of stateful NAT44 gateways, too.

B. Problems with Benchmarking Stateful NAT44 Gateways

1) Problems with the Feasibility of RFC Compliant Tests

As for benchmarking stateful NAT44 gateways, we are faced with multiple problems. RFC 2544 requires testing with bidirectional traffic, whereas RFC 4814 requires the usage of pseudorandom port numbers with uniform distribution from the above mentioned ranges.

It can be easily calculated that the number of potential source port number destination port number combinations is more than three billion and it means so many network flows, thus potential entries in the connection tracking table of the stateful NAT44 gateways. Therefore, literally following this requirement in the private to public direction could exhaust the capacity of the connection tracking table of the DUT.

As for sending traffic in the public to private direction using pseudorandom port numbers, it would result in sending a lot of frames that do not belong to any existing connection, thus the stateful NAT44 gateway would simply discard them and the throughput test would fail.

2) Problems with the Widely-used Measurements

Researchers were creative enough to accommodate to the limitation of NAT44 that connections may be initiated only from the private side. They put the iperf or D-ITG server on the public side and thus the measurement was feasible. However, this type of measurement has serious limitations. To examine them, let us see, what happens (and what may happen) during the execution of a test. At the beginning of the test, most of the test frames sent form the private side result in new connections in the stateful NAT44 gateway. As time elapses, its connection tracking table has more and more connections and thus the proportion of the test frames that belong to an existing connection will increase. The proportion of the test frames resulting in new connections will likely decrease significantly and it may even become zero. The progress of this change depends on several factors including:

1. how the client is programmed (how many different network flows are used and what policy it follows to send a test frame that belongs to an already used or a new flow)
2. the connection timeout time of the stateful NAT44 gateway
3. the size and policy of the connection tracking table of the stateful NAT44 gateway.

The main problem with this type of measurement is that the test traffic is a kind of a mix, in which the proportion of the ingredients (frames resulting in a new connection or not) varies with the time. It results in several negative consequences, including:

1. It is rather hard to tell, exactly what was measured: e.g., the connection setup performance or the frame forwarding performance of the DUT?
2. The results of measurements performed with different tools are likely not comparable.

3. It is hard to tell, what conditions are needed to achieve reproducible measurements.
4. It is not possible to measure some clear and well defined characteristics like bidirectional, download-only, and upload-only throughput or connection setup performance.

Our methodology offers remedy for all these problems.

C. *Our Methodology*

We have defined a general methodology [14] suitable for the benchmarking of any stateful NATxy gateways using RFC 4814 pseudorandom port numbers, where x and y are in {4, 6}.

Now we give a brief introduction to the methodology using the example of the stateful NAT44.

The test setup is shown in Fig. 2. The DUT is the stateful NAT44 gateway, which has a connection tracking table. Its content, size, and replacement policy is unknown for the Tester. The Tester can influence or examine its content in indirect ways:

- The Tester can add a new connection to the connection tracking table by sending a test frame in the private to public direction with a new source port number destination port number combination.
- The Tester can check, if a given connection is present in the connection tracking table by sending a test frame belonging to the given connection in the public to private direction and verifying if the test frame arrives back.

There are two operations that can be performed by some out of band methods:

1. The timeout time of the connections can be set to any permitted value.
2. The entire content of the connection tracking table can be deleted.

Please refer to Section V.A, how these operations can be performed with iptables.

As the operation of the stateful NAT44 gateway is asymmetric, the operation of the Tester is also asymmetric.

The Initiator can send a test frame using any desired source port number destination port number combinations, but it uses restricted ranges to avoid the exhaustion of the capacity of the connection tracking table of the DUT. The size of the source port number range is larger (e.g. a few times 10,000) and the size of the destination port numbers is smaller (e.g. in the order of 10, 100, or 1000), and it can be used as a parameter to perform the measurements with different number of network flows. Please refer to our Internet Draft [14] for the rationale of the asymmetry of the sizes of the port number ranges. (The source and destination IP addresses have constant values and the protocol is always UDP.)

The Responder may not invent any flow identifiers, but it extracts the *four tuples* (source IP address, source port number, destination IP address, destination port number) from the received test frames and stores them in its *state table*. When it sends a test frame, it takes a four tuple from its state table (swaps source and destination), and thus it creates a valid test frame that belongs to an existing connection in the connection tracking table of the DUT.

To make testing possible, we have introduced the *preliminary test phase*. During this phase only the Initiator sends test frames. The DUT registers the new connections into its connection tracking table, translates the test frames and forwards them to the Responder. Thus, the connection tracking table of the DUT and the state table of the Responder are initialized, and in the *real test phase*, the Responder is able to send valid test frames.

To achieve clear and repeatable measurements, we use two extreme situations that we can simply ensure:

1. All test frames create a new connection during the preliminary test phase.
2. Test frames never create a new connection in the real test phase.

We achieve them by using

- large enough and empty connection tracking table for each test
- pseudorandom enumeration of all possible source port number destination port number combinations in the preliminary test phase
- a properly high timeout value in the DUT (higher than the time duration from the beginning of the preliminary test phase to the end of the real test phase including timeout).

To quantify the connection setup performance of the DUT, we have introduced the *maximum connection establishment rate* as a new metric. It is the highest constant frame rate at which the DUT is able to process all test frames in the preliminary test phase. (Each test frame is successfully translated and a new connection is created in the connection tracking table.) Its measurement procedure is very simple to that of the throughput, the details can be found in our Internet Draft.

All "classic" measurements (throughput, latency, frame loss rate, etc.) can be performed in the real test phase. To that end, first, the preliminary test phase has to be executed using a frame rate safely lower than the measured connection establishment rate. Then comes the real test phase with the desired measurement.

The other side of connection establishment is connection tear down. We defined *connection tear down rate* to quantify the connection tear down performance of the DUT. Is short, it is measured as follows. First, N number of connections are loaded into the connection tracking table of the DUT. Then the entire content of the connection tracking table is deleted, and its T deletion time is measured. The connection tear down rate is calculated as: N/T. It is measured for different values of N.

We give more details in Section V.B.4, were we describe our test to measure the connection tear down rate of iptables.

IV. IMPORTANT DETAILS OF IPTABLES

First of all, iptables does *connection tracking* not only for stateful protocols (TCP), but also for stateless ones (UDP, ICMP). The connection tracking system of iptables is

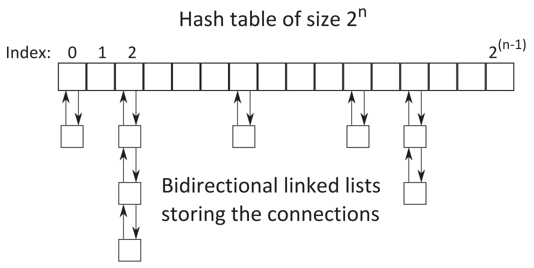

Fig. 3. Implementation of the connection tracking table of iptables using a hash table and bidirectional linked lists (based on [2]).

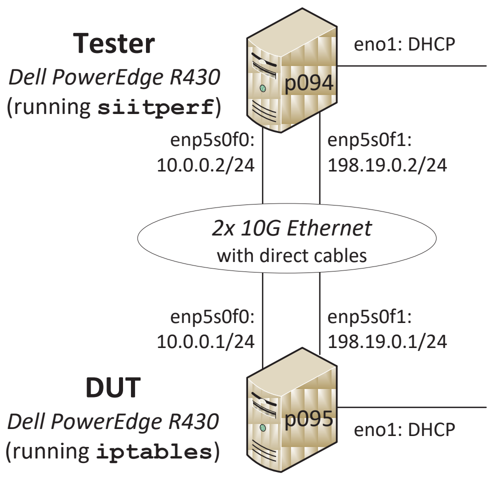

Fig. 4. Topology of the stateful NAT44 test system.

implemented using *hashing* to ensure efficient lookups. As for UDP, the source and destination IP addresses, as well as the source and destination port numbers are parts of the hash tuple. The size of the hash table is a power of 2. Hash collisions are handled that the *connection tracking entries* are stored as the elements of bidirectional linked lists starting from the *hash table* entries, as shown in Fig. 3.

A. Parameters to Tune

The *size of the hash table* fundamentally influences the efficiency of hashing and thus also the performance of iptables. Its default size is automatically determined on the basis of the memory size of the Linux system. It can be read or modified by reading or writing the **`/sys/module/nf_conntrack/parameters/hashsize`** file.

The *maximum number of the connection tracking entries* is another important parameter. It can be read or written using **`sysctl`** as **`net.netfilter.nf_conntrack_max`**.

Traditionally, the values of the above two parameters are set as: **`hashsize=nf_conntrack_max/8`** [15], [16]. It means that the *average* length of the linked list starting from the hash table entries may be up to 8. However, this is just a convention, and in our current paper, we examine what value is worth using.

We note that newer systems use 4 instead of 8 when they set the default values, please refer to Table VI.

As for timeout, iptables handles several different timeouts for the various states of TCP, two different ones for UDP and one for ICMP. Now for us it is enough to handle (using **`sysctl`**): **`net.netfilter.nf_conntrack_udp_timeout`**.

B. Memory Consumption

The memory consumption of the *hash table* is likely 8 bytes per entry, as 64 bits are required to store a pointer pointing to the first element of the linked list, and we also confirm it by measurements in Section V.C.1.

As for the memory consumption of the *connection tracking entries*, we have found several different values.

1. Section 3.7 of [17] (a document from 2007) states that "each tracked connection eats about 350 bytes of non-swappable kernel memory".
2. Another source from 2009 stated that as each object was 304 bytes in size and 13 of them fitted in a 4096 bytes page, thus 144 bytes was wasted and so the effective memory consumption was about 316 bytes per **`nf_conntrack`** object [18].
3. We found in a commit message from 2016 that "increases struct size by 32 bytes (288 -> 320), but it is the right thing, else any attempt to (re-)arrange nf_conn members by cacheline won't work." [19]

Therefore, we have determined the effective memory consumption of the *connection tracking entries* by measurement in Section V.C.2.

V. Our Benchmarking Measurements

A. Measurement System

The topology of the measurement system is shown in Fig. 4. Both the Tester and the DUT were Dell PowerEdge 430 servers. Each one had two 2.1GHz Intel Xeon E5-2683 v4 CPUs, 384GB 2400MHz DDR4 RAM and an Intel 10GbE dual port X540 network adapter. They were interconnected by direct cables to avoid frame loss. Debian 9.13 Linux operating system with 4.9.0-16-amd64 kernel was used on both servers. The CPU clock frequencies of both servers were set to fixed 2.1GHz using the **`tlp`** Debian package and hyper-threading was switched off to ensure reliable results. Only CPU cores 0-15 of the DUT were online. The version of iptables was 1.6.0. The test system was the same as we used for our measurements in Section 2 of [4], thus our results are directly comparable.

For performing measurements, we used the **`siitperf`** [20] RFC 8219 compliant SIIT and stateful NAT64 / NAT44 tester program documented in [21].

We note that **`siitperf`** was developed for research purposes and it is a collection of *binary programs* (written in C/C++ using DPDK) that perform elementary tests, and **`bash`** *shell scripts* that have to be tailored to the measurement environment. As for stateful tests, they use **`ssh`** to execute some scripts on the DUT e.g., to empty the connection tracking table of the DUT. For our current tests, we used two such scripts on the **`p095`** server. The **`set-iptables-hcpx`** script was used to set the iptables rule, the size of the hash table,

TABLE I
MAXIMUM CONNECTION ESTABLISHMENT RATE OF IPTABLES AS A FUNCTION OF THE HASH TABLE SIZE, 40M CONNECTIONS, 16 CPU CORES

Hash table size	2^26	2^25	2^24	2^23	2^22	2^21
Number of connections / hash table size	0.5960	1.1921	2.3842	4.7684	9.5367	19.0735
Error of binary search (cps)	100	100	100	100	100	100
Median (cps)	2,263,732	2,075,195	1,696,411	1,231,993	780,090	440,124
Minimum (cps)	2,203,063	2,044,493	1,624,938	1,216,734	764,098	421,813
Maximum (cps)	2,359,680	2,125,549	1,750,061	1,251,037	797,912	445,617
Median / previous median	-	0.92	0.82	0.73	0.63	0.56

TABLE II
THROUGHPUT OF IPTABLES AS A FUNCTION OF THE HASH TABLE SIZE, 40M CONNECTIONS, 16 CPU CORES, BIDIRECTIONAL TRAFFIC

Hash table size	2^26	2^25	2^24	2^23	2^22	2^21
Number of connections / hash table size	0.5960	1.1921	2.3842	4.7684	9.5367	19.0735
Error of binary search (fps)	200	200	200	200	200	200
Median (fps)	4,252,197	4,068,973	3,683,713	3,150,632	2,445,677	1,719,848
Minimum (fps)	4,174,436	3,899,872	3,624,876	2,999,876	2,428,586	1,716,672
Maximum (fps)	4,282,348	4,103,502	3,719,360	3,187,622	2,459,836	1,728,148
Median / previous median	-	0.96	0.91	0.86	0.78	0.70

TABLE III
CONNECTION TEAR DOWN RATE OF IPTABLES AS A FUNCTION OF THE HASH TABLE SIZE, 40M CONNECTIONS, 16 CPU CORES

Hash table size	2^26	2^25	2^24	2^23	2^22	2^21
Number of connections / hash table size	0.5960	1.1921	2.3842	4.7684	9.5367	19.0735
Filled table deletion time – Median (s)	158.51	154.07	153.52	150.61	150.12	149.76
Filled table deletion time – Minimum (s)	157.63	152.81	152.21	149.94	148.62	148.43
Filled table deletion time – Maximum (s)	158.89	154.72	153.69	151.50	150.41	150.13
Empty table deletion time – Median (s)	8.00	4.15	2.24	1.26	0.78	0.55
Empty table deletion time – Minimum (s)	7.99	4.14	2.22	1.25	0.77	0.54
Empty table deletion time – Maximum (s)	8.02	4.17	2.25	1.28	0.81	0.57
Net connection deletion time (s)	150.51	149.92	151.29	149.35	149.34	149.21
Connection tear down rate (cps)	265,763	266,809	264,402	267,827	267,845	268,079

the maximum number of connections, and the UDP timeout value before each elementary test. The **`del-iptables`** script deleted the iptables rule and the content of the connection tracking table (by the removal of the kernel modules) after each elementary test. They are available on GitHub [22].

B. *Performance Measurements*

1) *Aim, Parameters, and Types of Tests*

We aimed to examine, how the ratio of the number of connections and the hash table size influences the performance of iptables.

Gapon [16] recommended 4,194,304 as the upper limit for number of connections for a highly loaded NAT server and 524,288 for hash table size. We decided to use rather 40M connections, because we wanted to test iptables under really demanding condition. We achieved this number of port number combinations by using 40,000 source port numbers and 1,000 destination port numbers.

To be able to handle 40M connections, the first appropriate power of 2 for the maximum number of connections is 2^{26}=67,108,864. As for hash table size, first, we used the same value and then we halved it five times, thus the final tested value was 2^{21}=2,097,152.

We set the UDP timeout to 10,000 seconds, to guarantee its high enough value for all tests.

We measured the maximum connection establishment rate, the throughput, and connection tear down rate with each hash table size. All measurements were performed 10 times to get reliable results.

2) *Maximum Connection Establishment Rate*

The maximum connection establishment rate of iptables as a function of the hash table size is shown in Table I. (The "error of binary search" value expresses the stopping criterion for the binary search. It stops, when:

higher_limit – lower_limit <= error.)

Although the independent variable is the hash table size (shown as a power of 2), what really helps to understand the behavior of the system is the average number of connections hashed to the same hash table entry, that is *the average length of the linked lists*. It is computed as the number of connections per hash table size. In the first step, it increases from about 0.6 to about 1.2, and the median of the maximum connection establishment rate decreases only 8%. However, its further doubling causes more and more radical decrease of the median.

3) *Throughput*

The throughput of iptables as a function of the hash table size is shown in Table II. It behaves similarly to the maximum connection establishment rate in the sense that the doubling of the average length of the linked list causes more and more radial decrease of the performance when it becomes significantly higher than 1, but the measure of the deterioration is lower.

4) *Connection Tear Down Rate*

Having no better way to measure the connection tear down rate, we used an aggregate measurement that *N* number of connections were loaded into the connection tracking table of iptables and then the entire table was deleted and the *T* duration of the deletion was measured. However, then the deletion time

TABLE IV
MEMORY CONSUMPTION OF THE HASH TABLE OF IPTABLES AS A FUNCTION OF THE HASH TABLE SIZE

Hash table size	2^26	2^25	2^24	2^23	2^22	2^21
Median (kB)	523,560	261,514	130,238	64,964	32,444	16,248
Minimum (kB)	521,816	260,208	129,024	63,436	30,824	15,368
Maximum (kB)	527,092	263,284	132,108	66,108	33,132	16,896
Bytes per hash table entry	7.99	7.98	7.95	7.93	7.92	7.93

TABLE V
MEMORY CONSUMPTION OF IPTABLES WITH 40M CONNECTION TRACKING ENTRIES AS A FUNCTION OF THE HASH TABLE SIZE

Hash table size	2^26	2^25	2^24	2^23	2^22	2^21
Number of connections / hash table size	0.5960	1.1921	2.3842	4.7684	9.5367	19.0735
Median (kB)	15,056,120	15,056,038	15,056,018	15,055,478	15,053,256	15,054,764
Minimum (kB)	15,055,296	15,050,388	15,053,528	15,050,028	15,050,876	15,051,868
Maximum (kB)	15,058,672	15,057,412	15,060,400	15,057,376	15,057,096	15,057,196
Bytes per hash table entry	385.44	385.43	385.43	385.42	385.36	385.40

contained the time necessary to delete an empty connection tracking table, as well as the command execution and communications latencies, too. To make our results more accurate, we have also measured the duration of the deletion of an empty table, which also contained the command execution and communications latencies. Thus their difference contains only the time spent by the deletion of the *N* number of connections, which is called as *net connection deletion time* in Table III. This value is nearly the same independently form the hash table size. Thus, the connection tear down rate is also independent from the size of the hash table.

C. Memory Consumption Measurements

1) Hash Table

To measure the memory consumption of the hash table, we set various hash table sizes, and checked, how the memory usage of the Linux systems changed. We considered the "used" value in the output of the **`free`** Linux command. We could not set arbitrarily small hash table size: if we tried setting it to a smaller value than 512 entries, then it was set to 512.

As with the other measurements, we set each size 10 times (including 512) and recorded the amount of the used memory of the Linux system with a script. Then we subtracted the memory usage measured with 512 entries from all the other memory usage values. We calculated median, minimum and maximum of the results, and finally, we computed the memory consumption per hash table entry using the median values for the calculation. The results are shown in Table IV. Of course, the 512*8=4,096 bytes memory consumption of the hash table causes some small error, but the results can still confirm that the *memory consumption of the hash table is 8 bytes per entry*.

2) Connection Tracking Entries

To measure the memory consumption of the connection tracking entries, we inserted 40M connections into the connection tracking table using safely lower frame rates than the maximum connection establishment rate for the given hash table size. Then we recorded the memory usage of the Linux system, next, deleted the content of the connection tracking table, and finally, recorded the memory usage of the Linux system again.

As with the other measurements, we performed the tests with each connection tracking table size 10 times.

We calculated the difference of the memory usage of the Linux system when the connection tracking table had 40M entries and when it was empty. This difference is the memory consumption of the 40M connection tracking entries. The results are shown in Table V. The *memory consumption of the 40M connection tracking entries is independent from the size of the hash table*, and on average, a *single connection tracking entry consumes 385.4 bytes*. The results are very stable: the difference of the maximum and minimum is always less than 0.1% of the median.

VI. DISCUSSION OF THE RESULTS AND OUR RECOMMENDATION FOR SETTING HASH TABLE SIZE

Our performance measurements showed that value of the *number of connections per hash table size* is a very important parameter that highly influences the performance of iptables. This parameter gives the average length of the linked lists starting from the entries of the hash table. Both the maximum connection establishment and the throughput of iptables seriously deteriorates when this number becomes significantly higher than 1. But the connection tear down rate does not depend on it at all.

We have also checked the "price" of the performance and we found that the memory consumption of the hash table is proportional to its size: each entry requires 8 bytes. However, the memory consumption of the connection tracking entries does not depend on the size of the hash table, and each entry occupies approximately 385.4 bytes. The orders of magnitude of these two numbers suggest us that the memory consumption of the hash table entries is practically negligible compared to the memory consumption of the connection tracking entries. With other words: the 40M connection tracking entries occupy about 15GB RAM independently from the hash table size as shown in Table V, whereas the memory consumption of the hash table itself varies between 0.5GB and 16MB as shown in Table IV, thus the latter is practically negligible.

Therefore, we definitely recommend to abandon using the **`hashsize=nf_conntrack_max/8`** convention and rather use **`hashsize=nf_conntrack_max`** to increase the performance of iptables significantly. Of course, now arises the question of using **`hashsize=nf_conntrack_max*`*n***, where $n > 1$. We

TABLE VI
Default and Maximum Allowed Values for **hashsize** and **nf_conntrack_max** as a Function of Memory Size, Iptables 1.6.0

Computer memory size (GB)	1	2	8	384
Default **hashsize**	7,680	16,384=2^14	65,536=2^16	65,536=2^16
Default **nf_conntrack_max**	30,720	65,536=2^16	262,144=2^18	262,144=2^18
Maximum possible **hashsize**	33,554,432=2^25	67,108,864=2^26	268,435,456=2^28	268,435,456=2^28
Maximum possible **nf_conntrack_max**	1,073,741,824=2^30	1,073,741,824=2^30	1,073,741,824=2^30	1,073,741,824=2^30

TABLE VII
Performance of Nftables 0.9.0-2, 2^26 Hash Table Size, 40M Connections, 16 CPU Cores

Performance metric	Max. conn. est. rate (cps)	Throughput (fps)
Error of bin. search	100	200
Median	228,222	835,544
Minimum	225,683	824,804
Maximum	229,198	840,428

did not do tests with n=2, 4, 8, etc. but our results show that the expectable gain is much less. When using iptables as a stateful NAT44 gateway to forward Internet traffic, there are a high number of packets transferred per session. Thus, throughput is the dominant one from among our three used performance metrics. Examining Table II, we can see that there is only 4% performance difference between the first two columns. And considering the observable trends, it is likely to be even less, if the size if the hash table is further increased. And the increase of the hash table size has a built in limit. Theoretically, the default and the allowed maximum values for **hashsize** and **nf_conntrack_max** depend on the RAM size of the computer [15]. Our measurements show that it is true for very small RAM sizes (e.g. 1GB or 2GB), but the values do not change from 8GB and to 384GB RAM size. We used the **mem=*n*GB** kernel command line parameter to limit the available memory for testing. Our results in Table VI show that the default and maximum values are the same for 8GB and 384GB.

VII. Testing the Performance of Nftables

We have also tested the maximum connection establishment rate and throughput performance of nftables. We used the same test system as shown in Fig. 4, but Debian 10.13 with 4.19.0-20-amd64 kernel was used on the DUT. The version of nftables was: 0.9.0-2. We tested its performance only at the “optimal” working point, that is, using our recommended setting: **hashsize=nf_conntrack_max**.

The results are shown in Table VII. Comparing the results with that of iptables (shown in Table I and Table II), we can see that the median maximum connection establishment rate of nftables (228,222cps) is about one tenth of the median maximum connection establishment rate of iptables (2,263,732cps), whereas the median throughput of nftables (835,544fps) is about one fifth of the median throughput of iptables (4,252,197fps) measured under the same conditions. Therefore, we conclude that nftables may not replace iptables in the application scenarios where a stateful NAT44 gateway of a CGN system has to handle a high number of connections with high performance.

VIII. Conclusion

We have measured the maximum connection establishment rate, throughput and connection tear down rate as well as the memory consumption of iptables as a function of the hash table size using always 40 million connections to determine the optimal value for the ratio of the number of connections and the hash table size and/or any possible tradeoff.

We conclude that the long established convention of **hashsize=nf_conntrack_max/8** should be replaced by the **hashsize=nf_conntrack_max** rule to increase the performance of iptables to a high extent.

We have also shown that nftables may not replace iptables in the application scenarios where a stateful NAT44 gateway of a CGN system has to handle a high number of connections with high performance, because iptables achieved about ten times higher maximum connection establishment rate and about five times higher throughput than nftables.

Acknowledgment

The experiments were carried out by remotely using the resources of NICT StarBED, 2–12 Asahidai, Nomi-City, Ishikawa 923–1211, Japan.

The authors would like to thank Shuuhei Takimoto for the possibility to use StarBED, as well as to Tsukasa Nishita Makoto Yoshida for their help and advice in StarBED usage related issues.

Gábor Lencse thanks the National Institute of Information and Communications (NICT), Japan for their support of his stay at the Research Laboratory of Internet Initiative Japan, where his research topic was the performance analysis of stateful NAT64 / NAT44 implementations.

References

[1] M. Nikkhah, R. Guérin, “Migrating the Internet to IPv6: An exploration of the when and why”, *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 2291–2304, Apr. 2016, doi: 10.1109/TNET.2015.2453338

[2] P. N. Ayuso, “Netfilter's connection tracking system”, *Login: The Usenix Magazine*, vol. 31, no. 3, (2006) pp. 34–39. [Online]. Available: https://www.usenix.org/system/files/login/articles/892-neira.pdf

[3] D. Melkov, A. Šaltis and Š. Paulikas, “Performance Testing of Linux Firewalls”, *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, 2020, pp. 1–4, doi: 10.1109/eStream50540.2020.9108868.

[4] G. Lencse, “Scalability of IPv6 transition technologies for IPv4aaS”, Internet Draft, Oct 23, 2022, draft-lencse-v6ops-transition-scalability-04 [online], available: https://datatracker.ietf.org/doc/html/draft-lencse-v6ops-transition-scalability-04

[5] N.Gandotra, L.S. Sharma, “Performance evaluation and modelling of the Linux firewall under stress test”, In: Singh, P., Kar, A., Singh, Y., Kolekar, M., Tanwar, S. (eds) Proceedings of ICRIC 2019, *Lecture Notes in Electrical Engineering*, vol 597. Springer, DOI: 10.1007/978-3-030-29407-6_54

[6] K. Taga, J. Zheng, K. Mouri, S. Saito, E. Takimoto, “Firewall traversal method by pseudo-TCP encapsulation”, *IEICE Transactions on Information and Systems*, 2022, vol. E105.D, no. 1, pp. 105–115, 2022, DOI: 10.1587/transinf.2021EDP7050

[7] A. Toonk, “Linux kernel and measuring network throughput”, personal blog, [Online], available: https://toonk.io/linux-kernel-and-measuring-network-throughput/index.html

[8] M. Tumolo, “Towards a faster Iptables in eBPF”, MSc thesis, Politechnico di Torino, 2017-2018, [online], available: https://webthesis.biblio.polito.it/secure/8475/1/tesi.pdf

[9] S. Bradner, and J. McQuaid, “Benchmarking methodology for network interconnect devices”, *IETF RFC 2544*, 1999. DOI: 10.17487/RFC2544.

[10] D. Newman, T. Player, “Hash and stuffing: Overlooked factors in network device benchmarking”, *IETF RFC 4814*, 2008. DOI: 10.17487/RFC4814

[11] T. Herbert, W. de Bruijn, “Scaling in the Linux networking stack”, [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

[12] C. Popoviciu, A. Hamza, G. V. de Velde, and D. Dugatkin, “IPv6 benchmarking methodology for network interconnect devices”, *IETF RFC 5180*, 2008, DOI: 10.17487/RFC5180.

[13] M. Georgescu, L. Pislaru, and G. Lencse, “Benchmarking methodology for IPv6 transition technologies”, *IETF RFC 8219*, Aug. 2017, DOI: 10.17487/RFC8219

[14] G. Lencse, K. Shima, "Benchmarking methodology for stateful NATxy gateways using RFC 4814 pseudorandom port numbers", Internet Draft, Sep 24, 2022, draft-ietf-bmwg-benchmarking-stateful-00 [Online], available: https://datatracker.ietf.org/doc/html/draft-ietf-bmwg-benchmarking-stateful-00

[15] H. Eychenne, “Conntrack tuning: Netfilter conntrack performance tweaking, v0.8”, 2008, [Online], available: https://wiki.khnet.info/index.php/Conntrack_tuning

[16] V. Gapon, “Tuning nf_conntrack”, personal blog, [Online], available: https://ixnfo.com/en/tuning-nf_conntrack.html

[17] H. Welte, “Netfilter/iptables FAQ” 2007, [Online], available: https://www.netfilter.org/documentation/FAQ/netfilter-faq.html

[18] J. Leach, “Netfilter conntrack memory usage”, [Online], available: https://johnleach.co.uk/posts/2009/06/17/netfilter-conntrack-memory-usage/

[19] P. N. Ayuso, “[06/26] netfilter: conntrack: align nf_conn on cacheline boundary”, commit message, 2016, [Online], available: https://patchwork.ozlabs.org/project/netdev/patch/1467815048-2240-7-git-send-email-pablo@netfilter.org/

[20] G. Lencse, “Siitperf: an RFC 8219 compliant SIIT and stateful NAT64/NAT44 tester”, free software under GPLv3 license, source code, [Online], available: https://github.com/lencsegabor/siitperf

[21] G. Lencse, “Design and implementation of a software tester for benchmarking stateful NATxy gateways: theory and practice of extending siitperf for stateful tests”, *Computer Communications*, vol. 172, no. 1, pp. 75-88, Aug. 1, 2022, DOI: 10.1016/j.comcom.2022.05.028

[22] G. Lencse, “DUT settings for benchmarking iptables and nftables” [Online], available: https://github.com/lencsegabor/DUT-settings-iptables-nftables

Gábor Lencse received his M.Sc. and Ph.D. degrees in computer science from the Budapest University of Technology and Economics, Budapest, Hungary in 1994 and 2001, respectively.
He works for the Department of Telecommunications, Széchenyi István University, Győr, Hungary since 1997. Now, he is a Professor. He is also a part time Senior Research Fellow at the Department of Networked Systems and Services, Budapest University of Technology and Economics since 2005. His research interests include the performance and security analysis of IPv6 transition technologies. He is a co-author of RFC 8219 and RFC 9313.

Keiichi Shima is a deputy director at the Research Institute of Advanced Technology of SoftBank Corp. His research field is the Internet and mobile network, including designing and implementing communication protocols, operation technologies, network security, and so forth. He also works as a board member of the WIDE project operating a nation wide research network in Japan.