Making Stateless and Stateful Network Performance Measurements Unbiased

G. Lencse

Abstract—**The Benchmarking Working Group (BMWG) of the Internet Engineering Task Force (IETF) has defined a series of Requests for Comments (RFC) to standardize the benchmarking of network interconnect devices (e.g., bridges, routers, different IPv6 transition solutions). The paper points out that there are cases where the performance results are significantly different when a single IP address pair or multiple IP addresses are used. The cause of this phenomenon is rooted in the recent hardware and software advancements: Receive Side Scaling (RSS) makes it possible to distribute packet processing workload over multiple CPU cores. However, this may be implemented in two ways: the first way only includes the IP addresses into the hash function used to distribute the workload among the CPU cores, whereas the second one also includes the port numbers. RFC 4814 proposed an excellent solution for the second case by recommending the usage of pseudorandom port numbers during benchmarking; however, the first case was not handled properly, because no explicit recommendation was given regarding the usage of multiple IP addresses. This paper attempts to bridge this methodological gap; a practical solution is proposed for using pseudorandom IP addresses in various scenarios including the benchmarking of IPv4 and IPv6 routers and Network Address Translation from IPv6 Clients to IPv4 Servers (stateful NAT64) gateways. Its feasibility is shown by disclosing the details of its implementation in siitperf. Then the proposed solution is validated by both stateless and stateful tests. It is shown that the measurement results of the tests following the proposed solution can better characterize the true performance of the network interconnect devices that follow the first type of RSS implementation than the results of the tests using a single IP address pair.**

Index Terms—**network performance testers, receive side scaling, siitperf, throughput.**

I. INTRODUCTION

BENCHMARKING of network interconnect devices aims to accurately measure their certain standardized performance characteristics in order to obtain reasonable and comparable results, which are essential for both the developers and the users of the devices. To that end, the Benchmarking Working Group (BMWG) of the Internet Engineering Task Force (IETF) has defined a series of Requests for Comments (RFCs). RFC 2544 [1] was published in 1999, and it still determines how commercial network performance testers work. In its appendix, it has defined a *test frame format* with fixed IP addresses and fixed User Datagram Protocol (UDP) port numbers for router testing, which was very convenient for the manufactures of the testers, as the very same test frames could be reused. As time passed by, state-of-the-art routers started using multiple processing units, among which the network traffic was distributed by using the entropy provided by the different source and destination IP addresses and port numbers. This solution is called Receive-Side Scaling (RSS) [2]. To that end, RFC 4814 [3] highly recommends the use of pseudorandom port numbers during benchmarking, however, it did not provide a solution regarding the IP addresses for the general case (please refer to Section II.A.2 for the details).

Depending on the implementation, RSS may only include the source and destination IP addresses or it may also include the source and destination port numbers into the tuple used for hashing. RFC 4814 compliant testers work properly in the second case, however, pseudorandom port numbers cannot provide entropy if the *Device Under Test* (DUT) follows the first type of RSS implementation; therefore, these devices produce poor benchmarking results in RFC 4814 compliant laboratory tests, whereas they can exhibit a high performance in production environments where the usage of multiple IP addresses ensures the entropy for the proper operation of their RSS implementation. Therefore, the conditions of laboratory tests should be improved to ensure unbiased performance testing. To that end, this paper examines how the usage of multiple IP addresses can be introduced in the performance testing of various network interconnect devices. Practical recommendations are provided for the usage of pseudorandom source and destination IP addresses in the case of both stateless and stateful benchmarking following the approach of RFC 4814 regarding the port numbers. The most important design and implementation considerations for extending **`siitperf`** [4] to support the usage of multiple IP addresses are also disclosed. The solution proposed is validated by performing benchmarking measurements pointing out a significant improvement of the results. The findings should make an important contribution to the field of benchmarking network interconnect devices by making stateless and stateful performance measurements unbiased regarding the type of RSS implementation of the DUT.

The rest of this paper is organized as follows: Section II gives a summary of the background information regarding

Submitted: October 29, 2023.

G. Lencse is with the Department of Telecommunications, Széchenyi István University, Győr, H-9026, Hungary. (e-mail: lencse@sze.hu)

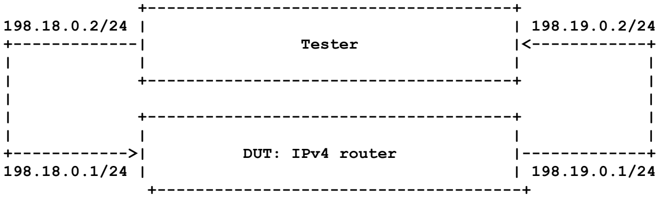

Fig. 1. Test setup for an IPv4 router (based on RFC 2544).

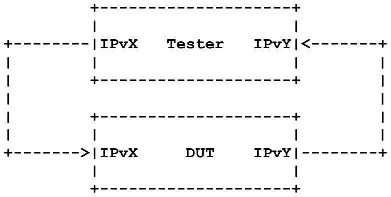

Fig. 2. Single DUT test setup [6].

stateless and stateful network interconnect device performance testing. Section III presents the recommendation for the introduction of multiple IP addresses. Section IV discloses the design and implementation of the extension of **siitperf** to support stateless and stateful benchmarking measurements with multiple IP addresses. In section V, the performance limits of the new functions of **siitperf** and its performance degradation due to the more complex operation are determined. The proposed methodology and its implementation are validated by stateless and stateful benchmarking measurements in various scenarios in Section VI and Section VII, respectively. The findings are discussed in Section VIII, and the paper is concluded in Section IX.

II. Introduction to the Benchmarking of Network Interconnect Devices

A. Benchmarking Methodology for IPv4 or IPv6 Routers

1) The original methodology

RFC 2544 has defined all the relevant aspects of benchmarking network interconnect devices including the *test and traffic setup*, standard *frame sizes*, *frame formats*, and *measurement procedures*. The primary recommended test setup is built up by two devices: the *Tester* and the *DUT*. Their corresponding network interfaces are connected and the Tester sends test frames through the DUT and receives back the frames, as shown in Fig. 1. It needs to be noted that although the arrows are unidirectional, *bidirectional traffic* is required. The essential benchmarking procedure is the *throughput measurement*, which determines the highest constant frame rate at which the DUT is able to forward all frames sent by the Tester. There are several other benchmarking procedures that give further insight into the performance of the DUT, like *latency* that determines the one way delay caused by the DUT measured at the frame rate previously determined by the throughput measurement procedure, or *frame loss rate*, which is to be determined at various frame rates. When routers are tested, it is required to perform the tests first, using a single source and destination IP address pair (as shown in Fig. 1) and then, using 256 different destination networks. The 198.18.0.0/15 IPv4 address range was reserved for benchmarking. Its first half (198.18.0.0/16) and its second half (198.19.0.0/16) are intended to be used on the left side and right side of the devices, respectively. Thus (numbering the bits from 0) the 16-23 bits ensure the possibility to describe the required 256 destination networks. As for the transport layer protocol, UDP was recommended.

It needs to be noted that benchmarking measurements are to be performed in an isolated laboratory environment and the usage of the dedicated address space can be a guarantee of preventing the measurement traffic from leaking out to the Internet.

2) Updates to the methodology

As time passed by, the methodology was updated in multiple ways.

As for router testing, RFC 4814 [3] requires the usage of pseudorandom source and destination port numbers from their specified ranges, 1,024–65,535 and 1–49,151, respectively. (If there is a requirement that either the source or the destination port number needs to have a specific value, then only the other port number should be pseudorandom.) However, section 4.4 of RFC 4814 considers the problem of IP addresses as solved in the general case. It only mentions the above-mentioned 8 bits (writes them as x.x.R.x/24) to be used as pseudorandom. However, this solution does not help, when the tests are done using a single destination network.

RFC 5180 [5] provided an update regarding the usage of IPv6 addresses. It has reserved a much larger address range for benchmarking: 2001:0:2::/48. However, it has explicitly declared *IPv6 transition technologies* out of its scope.

B. Benchmarking Methodology for Stateful NAT64 Gateways

1) The method in theory

RFC 8219 [6] has defined a comprehensive benchmarking methodology for IPv6 transition technologies. To that end, it classified the high number of IPv6 transition technologies [7] into a small number of categories (*dual stack*, *single translation*, *double translation*, and *encapsulation* technologies) regarding the solution used for packet traversal across the access and core network of the Internet Service Provider (ISP) and defined the test setup for each category. *Network Address Translation from IPv6 Clients to IPv4 Servers* (stateful NAT64) [8] belongs to the category of the single translation technologies. For this category, the *Single DUT test setup*, shown in Fig. 2, is recommended. It is similar to the test setup shown in Fig. 1, but here, different IP versions are used on the left side and on the right side of both the Tester and the DUT. Of course, both *X* and *Y* in *IPvX* and *IPvY* are from the set of {4, 6} and *X*≠*Y*.

As for the benchmarking procedures, RFC 8219 reused the throughput and the frame loss rate measurement procedures unchanged, it redefined the latency measurement procedure to provide more accurate results and added further procedures to

```
              +--------------------------------------+
    2001:2::2 |Initiator                    Responder| 198.19.0.2
+-------------|                Tester                |<------------+
| IPv6 address|                         [state table]| IPv4 address|
|             +--------------------------------------+             |
|                                                                  |
|             +--------------------------------------+             |
|   2001:2::1 |                DUT:                  | 198.19.0.1  |
+------------>|        Stateful NAT64 gateway        |-------------+
  IPv6 address|     [connection tracking table]      | IPv4 address
              +--------------------------------------+
```

Fig. 3. Test setup for benchmarking stateful NAT64 gateways. [9]

measure *Packet Delay Variation* (PDV) and *Inter Packet Delay Variation* (IPDV), whereas the latter was declared optional.

The requirement for benchmarking with bidirectional traffic was kept and benchmarking with unidirectional traffic was added as an optional test.

2) Practical problems

It turned out that benchmarking stateful NAT64 gateways requires further considerations because the verbatim application of certain requirements would result in various problems:

- The usage of pseudorandom source and destination port numbers in the IPv6 packets would result in potentially more than 3 billion connections, thus the test would exhaust the capacity of the connection tracking table of the stateful NAT64 gateway.
- The usage of pseudorandom port numbers in the IPv4 packets would result in packets that do not belong to any existing connection and the stateful NAT64 gateway would simply drop them.

Please refer to [9] for more details.

3) A solution to the problems

A general methodology suitable for the benchmarking of any *stateful NATxy gateways*, where *x* and *y* are in {4, 6}, using RFC 4814 pseudorandom port numbers was defined by this Internet Draft [10], which was adopted by the BMWG of IETF at the IETF 114 meeting on July 26, 2022.

A brief introduction to the benchmarking methodology for the stateful NAT64 gateways is provided by reusing the text of [9], [10] and [11].

The test setup is shown in Fig. 3. The DUT is the stateful NAT64 gateway, which has a *connection tracking table*.

The *Initiator* can send a test frame using any desired source port number destination and port number combinations, but it uses *limited ranges* to avoid the exhaustion of the capacity of the connection tracking table of the DUT. (According to the original methodology, only a single source and destination IP address pair was used [9].) Following the long established tradition of RFC 2544, RFC 5180, and RFC 8219 the UDP transport layer protocol is used.

The *Responder* receives the test frames and extracts the source IP address, source port number, destination IP address, destination port number (*four tuple*) from them then stores the four tuples in its *state table*. When it sends a test frame, it takes a four tuple from its state table (swaps source and destination), thus it creates a *valid test frame*, which belongs to an existing connection in the connection tracking table of the DUT.

The methodology uses two *test phases*. During *phase 1* 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, thus, in *phase 2*, the Responder is able to send valid test frames.

To achieve clear and repeatable measurements, two extreme situations are used:

1. During phase 1, all test frames create a new connection.
2. During phase 2, the test frames never create a new connection.

They can be simply ensured by using:

- a sufficiently large (to be able to store all the connections) and empty connection tracking table for each test
- pseudorandom enumeration of all possible four tuples in phase 1
- a properly high timeout value in the DUT (higher than the time duration from the beginning of phase 1 to the end of phase 2).

The *maximum connection establishment rate* has been introduced as a new metric to quantify the connection setup performance of the DUT. It is the highest constant frame rate at which the DUT is able to process all test frames in phase 1.

All "classic" measurements (throughput, latency, frame loss rate, etc.) can be performed in phase 2. To that end, first, phase 1 has to be executed using a frame rate safely lower than the measured connection establishment rate. Then comes phase 2 with the desired measurement.

The methodology was validated by performing its benchmarking measurements with three radically different stateful NAT64 gateway implementations [12].

C. Stateless Measurement Tools

Commercial network performance testers follow the requirements of RFC 2544 and the newer ones usually support the newer RFCs, too. In addition to this, they sometimes provide further optional features beyond the requirements of the RFCs. For example, it is quite common that they support non-zero loss throughput measurements, too. For example, the *Anritsu MP1590B* device has a parameter called *Loss Tolerance*. (Its value must be set to 0 to perform an RFC 2544 compliant throughout test.) However, it allows the user to set only a single IP address at each network port.

The *Spirent SPT-N4U* Tester also supports RFC 5180 tests for benchmarking IPv6 routers. With an appropriate trick, it was used for benchmarking various *stateless NAT64* implementations [13]. It has numerous advanced features, for example, when it is used in stateful mode, its *Avalance Commander* is able to generate IP addresses randomly from a specified range. However, when it is used in stateless mode for *Layer 2-3* tests including the RFC 2544 throughput measurements, it does not support using multiple IP addresses per its network ports, either.

`Siitperf` [4] is the world's first free software RFC 8219 compliant *Stateless IP/ICMP Translation* (SIIT) [14] (also called stateless NAT64) tester, written in C++ using Intel's *Data Plane Development Kit* (DPDK) [15] available from GitHub [16]. It was designed to be a flexible research tool and provides several features beyond the requirements of the RFCs, but it did not support the usage of multiple IP addresses prior to its current development, either.

D. *Stateful Measurement Tool*

As far as the author knows, the stateful extension of **`siitperf`** [9] is the only existing implementation of the concept for benchmarking stateful NATxy gateways using RFC 4814 pseudorandom port numbers described in [10]. It supports stateful NAT64 and stateful NAT44 measurements, but stateful NAT66 and stateful NAT46 measurements were not implemented. Its latest version prior to its current development only supported the use of a single source and destination IP address pair as documented in [9].

III. Recommendation for Using Multiple IP Addresses

The aim of the introduction of multiple IP addresses is the same as that of multiple port numbers, i.e. to support the even distribution of the load among multiple processing elements of network interconnect devices. To construct a similar solution to that of RFC 4814 regarding the port numbers, it was also considered to be desirable to use 16-bit address space. However, the size of the IPv4 address range reserved for benchmarking imposes a serious limitation. As for the stateless testing using IPv4 addresses, the author suggests two major solutions:

1. Only the last 8 bits of the IPv4 addresses are used. (The usable range is: 2-254, as 1 is used for addressing the DUT and 255 is the broadcast address.) Thus, it remains possible to use the 16-23 bits to describe 256 destination networks.
2. The last 16 bits of the IPv4 addresses are used. (The usable range is: 2-65,534.) Thus, the usage of 256 destination networks is sacrificed. This solution is shown in Fig. 4.

With regard to IPv6, there is no such problem, as the reserved benchmarking prefix contains an abundant number of bits. It even is possible to use exactly 65,536 different IPv6 address, as shown in Fig. 5. For simplicity, bits from 96 to 111 are used to distinguish 64k IPv6 addresses, and their last 16 bits are the same (expressing decimal 2). Bits from 56 to 63 can be used to describe the 256 destination networks.

When stateful NAT44 / NAT64 testing is designed, it should be considered that stateful NAT44 or NAT64 gateways that serve a high number of clients typically use more than a single public IPv4 address. However, in this case the entire 198.18.0.0/15 network can be used on the right side of the test setup, as shown in Fig. 6 and Fig. 7, because private IPv4 addresses or IPv6 address are used on the left side of the stateful NAT44 or stateful NAT64 gateway, respectively. (Due to the /15 mask, 198.18.255.255 and 198.19.0.0 are normal, usable IPv4 addresses.) It needs to be noted that the usage of 256 destination networks is out of the scope, as the performance of the stateful NAT44 / NAT64 gateway is measured and not the routing performance (even if the device also implements routing).

```
198.18.0.2/16-198.18.255.254/16      198.19.0.2/16-198.19.255.254/16
           \  +--------------------------------------+  /
            \ |                                      | /
+-------------|                Tester                |<------------+
|             |                                      |             |
|             +--------------------------------------+             |
|                                                                  |
|             +--------------------------------------+             |
|             |                                      |             |
+------------>|          DUT: IPv4 router            |-------------+
            / |                                      | \
           /  +--------------------------------------+  \
198.18.0.1/16                                        198.19.0.1/16
```

Fig. 4. Multiple IP address test setup for benchmarking IPv4 routers.

```
2001:2::[0000-ffff]:2/64             2001:2:0:8000::[0000-ffff]:2/64
           \  +--------------------------------------+  /
            \ |                                      | /
+-------------|                Tester                |<------------+
|             |                                      |             |
|             +--------------------------------------+             |
|                                                                  |
|             +--------------------------------------+             |
|             |                                      |             |
+------------>|          DUT: IPv6 router            |-------------+
            / |                                      | \
           /  +--------------------------------------+  \
   2001:2::1/64                                  2001:2:0:8000::1/64
```

Fig. 5. Multiple IP address test setup for benchmarking IPv6 routers.

```
10.[0-255].[0-255].2/8               198.19.0.0/15-198.19.255.254/15
           \  +--------------------------------------+  /
            \ |Initiator                    Responder| /
+-------------|                Tester                |<------------+
| private IPv4|                         [state table]| public IPv4 |
|             +--------------------------------------+             |
|                                                                  |
|             +--------------------------------------+             |
|  10.0.0.1/8 |                 DUT:                 | public IPv4 |
+------------>|        Stateful NAT44 gateway        |-------------+
  private IPv4|     [connection tracking table]      | \
              +--------------------------------------+  \
                                   198.18.0.1/15 - 198.18.255.255/15
```

Fig. 6. Multiple IP address test setup for benchmarking stateful NAT44 gateways.

```
2001:2::[0000-ffff]:2/64           198.19.0.0/15 - 198.19.255.254/15
           \  +--------------------------------------+  /
  IPv6      \ |Initiator                    Responder| /
+-------------|                Tester                |<------------+
| addresses   |                         [state table]| public IPv4 |
|             +--------------------------------------+             |
|                                                                  |
|             +--------------------------------------+             |
| 2001:2::1/64|                 DUT:                 | public IPv4 |
+------------>|        Stateful NAT64 gateway        |-------------+
 IPv6 address |     [connection tracking table]      | \
              +--------------------------------------+  \
                                   198.18.0.1/15 - 198.18.255.255/15
```

Fig. 7. Multiple IP address test setup for benchmarking stateful NAT64 gateways.

IV. Design and Implementation of Testing with Multiple IP Addresses

A. *Design Principles*

An existing software was to be extended, so the design considerations were the following:

1. To support flexible and convenient usage of multiple IP address during both stateless and stateful tests.
2. To fit together with the already existing design.
3. To facilitate a simple and efficient implementation.
4. To keep the performance of the Tester high.
5. To maintain the readability of the source code.

It needs to be noted that an introduction to **`siitperf`** is presented in the Appendix for the readers not familiar with it.

B. Parameter Design

To support flexibility and to follow the existing design, several new parameters were introduced.

The user should be able to decide if and how the IP addresses on the left side and right side should vary. The new parameters are: **`IP-{L,R}-var`**, and their possible values and meanings are:

0. use fixed IP addresses (as before)
1. increase the varying part of the IP addresses
2. decrease the varying part of the IP addresses
3. the varying part should be pseudorandom

It needs to be noted that if 0 is specified for both directions, the further parameters are completely redundant and fixed IP addresses are used.

The user should be able to specify the minimum and maximum values for the varying parts of the IPv4 or IPv6 addresses. The new parameters are: **`IP-{L|R}-{min,max}`**.

To support a simple and efficient implementation, the author decided to allow only 16 bits long varying part of both IPv4 and IPv6 addresses and to use the same parameters for IPv4 and IPv6 and for "real" and "virtual" addresses. (Please recall that the same code works with IPv4 and IPv6 when varying port numbers are used.) However, the *offset* of the 16-bit varying part (its distance from the beginning of the IP address) can be specified by the user independently for the two IP versions and for the left and right side addresses using the **`IPv{4,6}-{L,R}-offset`** parameters. Their valid range for IPv4 and IPv6 are 1-2 and 8-14, respectively. (They were later restricted due to simple implementation and performance considerations; please refer to Section V.F.1.)

The enumeration of the IP addresses can be controlled by the **`Enumerate-ips`** parameter. Its possible values and their meanings are:

0. no IP address enumeration
1. enumerate IP addresses in increasing order
2. enumerate IP addresses in decreasing order
3. enumerate IP addresses in pseudorandom order

It needs to be noted that the enumeration of the IP addresses may happen only in the first phase of stateful tests (similarly to the port number enumeration).

It also needs to be noted that the parameter design is validated at the end of the Appendix, where it is shown that the new parameters are suitable to express the settings required for the proposed test setups.

C. Implementation of the Stateless Measurements

Implementing the usage of multiple IP addresses for stateless tests was straightforward. The only important design decision worth mentioning was the introduction of a new **`msend()`** function to implement the usage of multiple IP addresses. Its rationale was to avoid the further increase of the number of sending loops in the **`send()`** function and thus to maintain the readability of the source code. As the support for multiple destination networks was sacrificed (to have enough bits to express multiple IP addresses) this new sender function has only two sending loops: one for using only multiple IP addresses but fixed port numbers and the other one for using both multiple IP addresses and multiple port numbers.

Otherwise, the same programming style was used as in the original code; the given fields of pre-generated templates were modified in the sending loops. To that end, pointers were set to the appropriate fields and more or less the same code was executed for the IPv4 and IPv6 test frames with the following two differences:

1. As opposed to IPv4 packets, IPv6 packets do not have header checksum. (Technically, the value of the field pointed by the corresponding pointer is set only if the IP version for the given side is 4 and the frame belongs to the foreground traffic.)
2. UDP checksum is mandatory for IPv6 packets, but it is optional for IPv4 packets. In the case of IPv4 packets, the 0 value of the field indicates that UDP checksum is not used. Therefore, if the checksum calculation results in a 0 value, its unary complement (0xffff) has to be written into the field. (As above, unary complement is used if the IP version for the given side is 4 and the frame belongs to the foreground traffic.)

D. Implementation of the Stateful Measurements

Whereas the IP addresses and port numbers can be handled independently from each other in the case of stateless measurements, the situation is rather different in the case of stateful measurements. The purpose of their enumeration is to use up all their possible combinations in phase 1 one so that no new four tuples (network flows) may appear in phase 2. To that end, both IP addresses and port numbers must be enumerated, except for the case when one of them has fixed values. So far, IP addresses had fixed values and only the port numbers could be enumerated [9] and under these conditions stateful benchmarking worked perfectly [12]. Similarly, if the port numbers have fixed values, it is enough to enumerate only the IP addresses.

The implementation of their enumerations required some careful considerations because the four possible values of the **`Enumerate-ports`** parameter and the four possible values of the **`Enumerate-ips`** parameter could have potentially 16 combinations. Their handling could have been implemented, for example, by a C language **`switch`** that has 16 **`case`**-es. However, it would have been only the high level structure of the program as both port numbers and IP addresses may be varying only partially (either the source or the destination, whereas the other one is fixed). As the author did not see much point in writing such a complicated program, the number of combinations to be implemented was reduced. The author believes that the following rule allows all practically

TABLE I
ALLOWED COMBINATIONS OF IP ADDRESS AND PORT NUMBER ENUMERATIONS AND HOW THEY ARE HANDLED

Enumerate-ports (right)	0	1	2	3
Enumerate-ips (below)				
0	These cases have already been implemented by the original **isend()** function.			
1	**imsend()** loop1	**imsend()** loop2	--	--
2	**imsend()** loop1	--	**imsend()** loop2	--
3	**imsend()** loop1	--	--	**imsend()** loop2

```
10.0.0.2/16 - 10.0.255.254/16       198.19.0.0/15 - 198.19.255.254/15
           \  +---------------------------------------+  /
            \ |Initiator                     Responder| /
+-------------|                Tester                 |<------------+
| private IPv4|                          [state table]| public IPv4 |
|             +---------------------------------------+             |
|                                                                   |
|             +---------------------------------------+             |
| 10.0.0.1/16 |                 DUT:                  | public IPv4 |
+------------>|        Stateful NAT44 gateway         |-------------+
 private IPv4 |     [connection tracking table]       | \
              +---------------------------------------+  \
                                   198.18.0.1/15 - 198.18.255.255/15
```

Fig. 8. Multiple IP address test setup for benchmarking stateful NAT44 gateways (modified version).

useful combinations: if any of the two parameters has the value of 0, then the other one may have any value, but if they both have non-zero values, then they must have the same value.

As for implementing the Initiator, the original **isend()** function was kept to handle the cases when **Enumerate-ips** has 0 value. Otherwise, the new **imsend()** function is used, which can be considered the *Initiator extension* of the new **msend()** function; it implements *IP address enumeration*. It has two sending loops: the first one handles the case when only the IP addresses are enumerated but the port numbers are not, and the second one handles the case when both the IP addresses and the port numbers are enumerated. Table I shows the summary of the allowed combinations of IP address and port number enumerations and how they are handled.

As stated above, the **imsend()** function was derived from the **msend()** function by adding IP address enumeration to it. An alternative could have been an "**misend()**" function that could have been derived from the **isend()** function by adding the usage of multiple IP addresses to it. However, that would have involved conflicts e.g., regarding the usage of the same bits for multiple destination networks and multiple IP addresses.

The new **imsend()** function kept the resilience of the **isend()** function in the sense that it supports port number enumeration but does not mandate it. When linear enumeration of the IP addresses and port numbers is done, they act as a four times two-byte counter, but the four two-byte fields of the counter may only take the values allowed by their specified ranges. (The 16-bit field in the destination IP address is the most significant one, then comes the 16-bit field in the source IP address, next the destination port number, and finally, the source port number is the least significant one.) Pseudorandom enumeration was implemented in the same way as before by using pre-generated values.

It needs to be noted that no modifications to the **rreceive()** and **rsend()** functions were needed because they both handle the full four tuple.

E. Restrictions due to Implementation Considerations

Two changes were made to the design at the implementation stage due to considerations of checksum calculation.

1) IPv4 and IPv6 offset

Both IPv4 header checksum and UDP checksum are calculated on 16 bits, that is two bytes. Using odd numbers as **IPv{4,6}-{L,R}-offset** would complicate the checksum calculation; therefore, the author decided to allow only even numbers. This constraint is not significant in the case of IPv6. As for IPv4, it means that the only allowed offset is 2. This restriction influences the test setup shown in Fig. 6. It was modified, as shown in Fig. 8. The loss of a few IP addresses from 65,536 is absolutely negligible.

Thus, the IPv4 offset parameters could have been fully eliminated, but they were still kept as they might be useful later if an unforeseen application scenario requires allowing for (and implement) further offset value(s).

2) IP address enumeration versus fixed IP addresses

As stated before, when the value of the **Enumerate-ports** parameter required port number enumeration, it was done in phase 1, even if the port numbers did not change in phase 2 due to the values of the **{Fwd,Rev}-var-{s,d}port** parameters. This approach could have been followed with regard to the IP addresses just for consistency. However, this approach would lead to certain contradiction during the implementation. To be able to handle the checksums correctly, the changing parts of the IP addresses are masked to 0, if the values of the appropriate **IP-{L|R}-var** parameters are non-zero. (All IP addresses exist only in a single copy in the **Throughput** class.) How should they be handled, if the **Enumerate-ips** parameter requires IP address enumeration, but the values of the **IP-{L|R}-var** parameters are 0?

- If they are not masked to 0, then their checksum will not be correct in phase 1.
- If they are masked to 0, then those bits are lost and will not be available in phase 2.

Of course, two copies of the IP addresses could have been used, but it would require a significant modification of the existing code and the author considered that it was not worth the effort, as he did not see any reasonable application scenario of using IP address enumeration in phase 1 and then

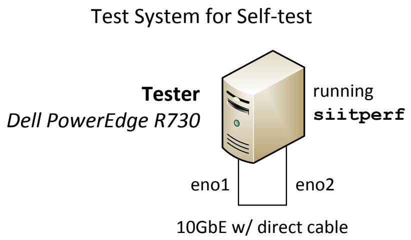

Fig. 9. Test system for the self-test of the Tester.

using fixed IP addresses in phase 2 of the stateful tests. Therefore, the author has come to the conclusion that the non-zero value of the **Enumerate-ips** parameter requires the **IP-{L|R}-var** parameters not to have zero values, either. (The program checks it and gives an appropriate "Input Error" message if the condition is not met.)

TABLE II
STATELESS SELF-TESTS: IPV4 THROUGHPUT OF THE TESTER @ 3.2GHZ

IP addresses port numbers	fixed fixed	fixed random	random fixed	random random
Median (fps)	7,161,269	7,184,723	7,175,444	7,161,422
Minimum (fps)	7,124,510	7,122,922	7,116,697	7,108,397
Maximum (fps)	7,226,746	7,214,859	7,219,178	7,203,141
Dispersion (%)	1.43	1.28	1.43	1.32

TABLE III
STATELESS SELF-TESTS: IPV6 THROUGHPUT OF THE TESTER @ 3.2GHZ

IP addresses port numbers	fixed fixed	fixed random	random fixed	random random
Median (fps)	7,168,274	7,193,027	7,189,696	7,122,892
Minimum (fps)	7,139,500	7,181,418	7,150,621	7,108,885
Maximum (fps)	7,208,871	7,257,874	7,250,252	7,207,073
Dispersion (%)	0.97	1.06	1.39	1.38

V. SELF-TEST OF THE TESTER

A. *Meaning and Purposes of the Self-Tests*

The self-test of the Tester means that its two interfaces are connected omitting the DUT, and benchmarking measurements are performed. The achieved frame rate of the throughput test depends on the frame sending and frame receiving abilities of the Tester. The self-test is a must before using the Tester for measurements, otherwise, there is no guarantee that the Tester measures the performance of the DUT; the Tester itself may be the bottleneck. In addition to this the self-test can be used to measure the performance penalty of the new functions and it raises the question to what extent the performance of the Tester decreased due to the new functions?

B. *Test System for the Self-Test of the Tester*

The topology of Test System for self-test is shown in Fig. 9. The most important parameters of the used Dell PowerEdge R730 server were: two 3.2GHz Intel Xeon E5-2667 v3 CPUs having 8 cores each, 8x16GB 2133MHz DDR4 SDRAM (accessed quad channel), and Intel X540-T2 10GbE network adapter. Hyper-threading was switched off and the CPU clock frequency was set to 3.2GHz (fixed), the nominal clock frequency of the CPU, to achieve stable measurement results using the **tlp** Linux package. Later it was set to 1.2GHz, the minimum clock frequency of the CPU, in the same way. Debian Linux 9.13 operating system with 4.9.0-13-amd64 kernel and DPDK 16.11.11-1+deb9u2 amd64 were used. As **siitperf** does not have version numbers, its version is identified by its latest commit 165cb7f on September 6, 2023.

C. *Stateless Tests and Results*

1) *Tests for guaranteeing the performance of the Tester*

To support stateless tests, IPv4 and IPv6 throughput tests were performed as self-test. The applied frame size was 64 bytes and 84 bytes for IPv4 and IPv6, respectively. As for the further parameters, all the four combinations of fixed and pseudorandom port numbers and IP addresses were used. The ranges for pseudorandom port numbers were those recommended by RFC 4814. The ranges for the varying part of the IPv4 and IPv6 addresses were 2-65,534 and 0-65,535, respectively. The parameters complied with the test cases are shown in Fig. 4 and Fig. 5. As required by RFC 2544 and its successors, bidirectional traffic was used. It needs to be noted that **siitperf** reports the results as frames/second *per direction*. It means that in all, **siitperf** sent and received twice as many frames as reported due to the bidirectional traffic.

The clock frequency of the Tester was set to fixed 3.2GHz, the nominal clock frequency of the CPU, to achieve a high performance. The results of the IPv4 and IPv6 throughput tests are shown in Table II and Table III, respectively. All measurements were executed 10 times and median was used as summarizing function and the minimum and maximum values were also included. The *dispersion* in the last line of the tables was calculated as follows:

$$dispersion = \frac{maximum - minimum}{median} \cdot 100\% \tag{1}$$

Regarding the guaranteed performance of the tester, the minimum values should be considered. In all cases, more than 7.1Mfps per direction frame rate was achieved. The bottleneck was always the X540 NIC as it is proven by the next series of measurements.

It can be observed that the dispersion of the results is always in the order of 1% magnitude. (It is used as a basis for comparison below.)

2) *Tests for checking the performance costs of operation*

To make the CPU performance the bottleneck, the clock frequency of the Tester was set to 1.2GHz, its lowest possible value. Otherwise, the same parameters were used as before. The results of the IPv4 and IPv6 throughput tests are shown in Table IV and Table V, respectively. The last rows of the tables show the relative performance of the given cases compared to the case when fixed IP addresses and fixed port numbers were used; the figures clearly show the performance penalty of generating pseudorandom port numbers or IP address or both. The IP version does not count for much, all four performance results are very similar to one another both in the case of IPv4

TABLE IV
STATELESS SELF-TESTS: IPV4 THROUGHPUT OF THE TESTER @ 1.2GHZ

IP addresses	fixed	fixed	random	random
port numbers	fixed	random	fixed	random
Median (fps)	7,026,719	4,293,187	4,089,050	2,828,910
Minimum (fps)	6,995,938	4,284,162	4,088,918	2,827,513
Maximum (fps)	7,062,523	4,294,927	4,089,133	2,830,567
Dispersion	0.95	0.25	0.01	0.11
Rel. perf. (%)	reference	61.10	58.19	40.26

TABLE V
STATELESS SELF-TESTS: IPV6 THROUGHPUT OF THE TESTER @ 1.2GHZ

IP addresses	fixed	fixed	random	random
port numbers	fixed	random	fixed	random
Median (fps)	7,049,596	4,297,380	4,138,784	2,863,442
Minimum (fps)	6,998,045	4,295,897	4,138,173	2,862,604
Maximum (fps)	7,078,370	4,297,495	4,140,626	2,864,345
Dispersion	1.14	0.04	0.06	0.06
Rel. perf. (%)	reference	60.96	58.71	40.62

TABLE VI
STATEFUL SELF-TESTS: MAXIMUM CONNECTION ESTABLISHMENT RATE TESTS, 4M CONNECTIONS; UNIDIRECTIONAL THROUGHPUT IN THE FORWARD DIRECTION USING FIXED FRAME FORMAT AS REFERENCE, TESTER @ 3.2GHZ

	In phase 1: pseudorandom enumeration of			Forward
IP addresses	--	4,000*1,0000	10*10*	throughput
port numbers	40,000*100	--	*400*100	fixed fr.
Median (fps)	10,069,082	9,944,307	9,936,514	11,878,250
Minimum (fps)	10,066,405	9,944,090	9,933,592	11,878,111
Maximum (fps)	10,069,227	9,944,347	9,937,012	11,878,925
Dispersion	0.03	0.00	0.03	0.01
Rel. perf. (%)	84.77	83.72	83.65	reference

and IPv6. Their performance is somewhat higher when only the port numbers are pseudorandom rather than in the case when only the IP addresses are pseudorandom. The explanation for this phenomenon is that varying IP addresses also influence the UDP checksum through the pseudo-header, but varying port numbers do not influence the IP header checksum. When fixed port numbers but pseudorandom IP addresses are used, the median IPv6 throughput (4,138,784fps) is somewhat higher than the median IPv4 throughput (4,089,050fps). The root cause for this is an extra conditional instruction in the source code handling IPv4 test frames: if the computed value of the UDP checksum happens to be 0, then 0xffff has to be stored.

It can be observed that the results using fixed frame format are still around 7Mfps and the dispersion of these results is also in the order of 1% magnitude. The dispersion of all the other results is at most 0.25%. The author assumes that the relatively higher dispersion of the results close to 7Mfps was caused by the race condition of the two senders that were competing for the limited resources of the same NIC. (The measurement log files show that the tests failed because one of the senders was not able to complete the sending of the required number of frames within 60.0006s.)

D. *Stateful Tests and Results*

As **siitperf** implements only stateful NAT64/NAT44 tests and stateful NAT66/NAT46 are out of its scope, its Receiver can handle only IPv4 packets. Therefore, when stateful self-tests are performed, the Initiator must send IPv4 packets (as the loopback wire does not do the stateful NAT64 translation, which is normally done by the omitted DUT).

However, considering that practically the same code modifies the IPv6 and IPv4 packet templates[1] and also the self-test results of the stateless test, one can easily infer that the results of the stateful NAT44 self-tests provide good guidelines for the stateful NAT64 benchmarking performance of the Tester, too.

First, it is considered, what kinds of measurements are needed to guarantee the performance of the Tester for stateful measurements.

In phase 1, four different cases are possible regarding the fixed or varying nature of the IP addresses and port numbers:

1. Only a single test frame is sent.
2. A single IP address pair and multiple port number combinations are used.
3. Fixed port numbers and multiple IP addresses are used.
4. Multiple IP addresses and multiple port numbers are used.

Based on the above four cases, the relevant benchmarking methodology [10] uses no. 2 and no. 4 with the conditions that in case 2, all possible source port number and destination port number combination must be enumerated in pseudorandom order, and in case 4, all the possible four tuples must be enumerated in pseudorandom order. As for case 1, although it is supported by **siitperf**, and the situation that all frames belonging to a single connection can be used as a reference measurement in phase 2, but there is no point in measuring the frame sending performance of **siitperf** when it sends only a single test frame in phase 1 as there is a gap between the two phases. Case 3 was considered interesting and it was included in the tests. (This case may be relevant when only a low number of connections are needed, but the aim is to have as many different IP addresses as possible.)

In phase 2, the Initiator uses the same sending functions (**send()** and **msend()**) as for stateless test, thus they require no further testing. And the Responder does not need testing either as it is implemented by the old **rsend()** function, which handles the full four tuple.

For the actual tests, 4 million connections were used as recommended by Gapon for a highly loaded NAT server [17]. In case 2, they are generated by using source port numbers 1-40,000 and destination port numbers 1-100, in case 3, using the 16-bit source address parts from 2-4001, and the 16-bit destination address parts from 2-1001, finally in case 4, the 16-bit source address parts from 2-11, the 16-bit destination address parts from 2-11, source port numbers 1-400 and destination port numbers 1-100.

The results are shown in Table VI. The results in the first

[1] With two exceptions: in the case of IPv4 test frames the IPv4 header checksum has to be set and the value of the UDP checksum has to be examined if it is 0 and if so, then 0xffff has to be used instead.

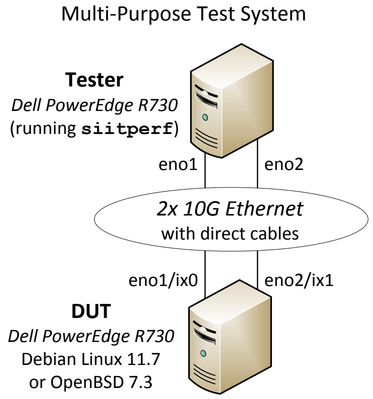

Fig. 10. Test system for the stateless and stateful tests both under Linux and under OpenBSD.

TABLE VII
IPv4 Packet Forwarding Performance of the DUT under Linux

IP addresses port numbers	fixed fixed	fixed random	random fixed	random random
Median (fps)	963,365	4,276,009	4,250,979	4,249,162
Minimum (fps)	959,051	4,275,288	4,249,983	4,248,696
Maximum (fps)	964,912	4,277,649	4,251,957	4,250,490
Dispersion (%)	0.61	0.06	0.05	0.04

TABLE VIII
IPv6 Packet Forwarding Performance of the DUT under Linux

IP addresses port numbers	fixed fixed	fixed random	random fixed	random random
Median (fps)	920,647	4,246,709	4,217,141	4,215,284
Minimum (fps)	919,785	4,245,603	4,216,513	4,213,819
Maximum (fps)	922,157	4,250,001	4,218,788	4,217,775
Dispersion (%)	0.26	0.10	0.05	0.09

three columns show the maximum connection establishment rates. They can be significantly higher than those in Table II because they were measured with unidirectional traffic; only the Initiator sent test frames to the Responder in phase 1. The last column shows throughput results measured with unidirectional traffic in the forward direction using fixed IP addresses and port numbers. It was added as reference to be able to see the cost of the pseudorandom enumeration of port numbers, IP addresses, or both. As the pseudorandom numbers are pre-generated, and they are read linearly from array(s) during the tests, there is much less performance penalty than in the case of the stateless tests when the pseudorandom numbers are generated during the test. As experienced with the stateless tests, the modification of the IP address is more "expensive" than that of the port numbers.

VI. Stateless Benchmarking Measurements

A. *Aim and Test System*

The aim of the stateless benchmarking tests was to examine if the usage of multiple IP addresses makes any difference in the throughput of IPv4 and IPv6 packet forwarding. To that end, the corresponding network interfaces of the Tester and that of the DUT were connected with direct cables, as shown in Fig. 10. The DUT had exactly the same hardware parameters as the Tester. The same test system was used for stateless and stateful measurements and the DUT was used with both Debian Linux 11.7 (with 5.10.0-23-amd64 kernel) and OpenBSD 7.3 (with GENERIC.MP #1125 kernel) operating systems. The CPU clock frequency was always set to fixed 3.2GHz under Linux, but the author could not set the CPU clock frequency to a fixed value under OpenBSD.

B. *Packet Forwarding Performance under Linux*

Basically, the same four types of tests were performed as in the case of the self-test: fixed frame format, only pseudorandom port numbers, only pseudorandom IP addresses, pseudorandom IP addresses and pseudorandom port numbers. However, it was experienced during the preliminary tests that using a high number of IP addresses significantly deteriorated the performance of the system. Therefore, only 1000 different IP addresses were used on each side. Regarding IPv4, it meant that the last two bytes of the IP addresses had the values from 2 to 1001. Regarding IPv6, the 12-13 bytes had the values from 0 to 999. The *Address Resolution Protocol* (ARP) and *Neighbor Discovery Protocol* (NDP) table entries were set manually in the DUT as **siitperf** was not able to answer ARP or NDP requests. When pseudorandom port numbers were used, *Receive Side Scaling* (RSS) was set in such a way that also port numbers may take part in the hash function using one of the four possible commands that can be expressed by the brace expansions below:

```
ethtool -N eno{1,2} rx-flow-hash udp{4,6} sdfn
```

When fixed port numbers were used, only the source and destination IP addresses took part in the hash function. To that end, only "**sd**" was used instead of "**sdfn**" in the commands above.

The results of the IPv4 throughput tests are shown in Table VII. When fixed frame format was used, the median frame forwarding rate was 963,365fps per direction. In this case only two CPU cores handled all interrupts (one CPU core per direction). In the other three cases the interrupts were distributed among all CPU cores. The results of the three cases were similar, but the difference was noticeable, when only the port numbers were pseudorandom, the median was 4,276,009fps, but when only the IP addresses were pseudorandom, the median was only 4,250,979fps. When the IP addresses were pseudorandom, then the usage of fixed or pseudorandom port numbers caused no significant difference (4,250,979fps vs. 4,249,162fps).

In addition to the above measurements, the test with fixed port numbers and pseudorandom IP addresses was also performed using 100 different IP addresses at each side. The median value of the throughput was 4,261,588fps per direction, which is still significantly lower than that with fixed IP addresses and pseudorandom port numbers. This difference shows that the usage of multiple ARP table entries has its performance costs.

The results of the IPv6 throughput tests are shown in Table VIII. Exactly the same tendencies can be observed as with

TABLE IX
IPv4 PACKET FORWARDING PERFORMANCE OF THE DUT UNDER OPENBSD

IP addresses port numbers	fixed fixed	fixed random	random fixed	random random
Median (fps)	390,125	384,596	1,277,414	1,283,352
Minimum (fps)	367,116	374,872	1,249,999	1,276,078
Maximum (fps)	437,745	441,549	1,296,876	1,297,120
Dispersion (%)	18.10	17.34	3.67	1.64

TABLE X
IPv6 PACKET FORWARDING PERFORMANCE OF THE DUT UNDER OPENBSD

IP addresses port numbers	fixed fixed	fixed random	random fixed	random random
Median (fps)	384,970	384,859	582,165	580,394
Minimum (fps)	351,553	382,807	577,024	562,499
Maximum (fps)	385,749	385,391	597,657	602,539
Dispersion (%)	8.88	0.67	3.54	6.90

IPv4 in Table VII. (The values themselves are slightly lower.)

C. *Packet Forwarding Performance under OpenBSD*

As the Packet Filter (PF) is enabled under OpenBSD 7.3 by default and its state handling has a significant impact on the performance of packet forwarding, PF was disabled manually using the **`pfctl -d`** command.

The same four types of tests were performed as in the case of Linux, and–as far as it was possible–the same parameters were used. However, OpenBSD does not support the setting of the parameters of RSS [18], and thus only the source and destination IP addresses took part in the hash function, the source and destination port numbers were ignored throughout the measurement process in each case.

The results of the IPv4 throughput tests are shown in Table IX. As expected, the usage of pseudorandom port numbers caused no significant performance difference compared to the cases when fixed port numbers were used; however, the usage of pseudorandom IP addresses resulted in a highly significant (more than 3-fold) performance increase compared to the cases when fixed IP addresses were used. The load conditions of the CPU cores were checked and it was found that the usage of pseudorandom IP addresses distributed the interrupts of packet arrivals more or less equally to each of the 16 CPU cores. However, significant systems load could be observed only on five of the CPU cores (numbered by OpenBSD as CPU01 to CPU05). The high dispersion of the results in the cases when fixed IP addresses were used may be explained by the fact that the interrupts were hashed to one of the CPU cores that was also used by the packet forwarding process; they competed for the usage of the CPU core, and their load was also changing with time.

The results of the IPv6 throughput tests are shown in Table X. The same tendencies can be observed in the case of IPv4, but in this case, the increase caused by the usage of multiple IP addresses was much lower.

VII. STATEFUL BENCHMARKING MEASUREMENTS

A. *Aim and Test System*

As in the stateless case, the aim of the stateful tests was to examine if the usage of multiple IP addresses makes any difference in stateful NAT64 benchmarking results.

B. *Stateful NAT64 Tests under Linux using Jool*

The version of Jool was 4.1.7, the *most mature version of Jool*. As far as the details of the measurements are concerned, the same method and even the same scripts were used as described in the Appendix of [12].

For all the three types of measurements, 4,000,000 connections were used, but they were achieved in three different ways:

1. Only the port numbers varied, the source port number range was: 1-40,000; the destination port number range was: 1-100.
2. Only the IP addresses varied, and the varying part of the IPv6 addresses took the values 0-3999, the varying part of the IPv4 addresses took the values 2-1001.
3. Both IP addresses and port numbers varied, and the varying part of the IPv6 address took the values 0-9, the varying part of the IPv4 address took the values 2-11, the source port number range was 1-400, and destination port number range was 1-100.

It needs to be noted that "source" and "destination" port numbers, as well as their ranges should always be interpreted in the traffic from the Initiator to the DUT. The stateful NAT64 gateway may change the source port numbers, and the Responder stores the four tuples received into its state table and it generates traffic only in phase 2, using the four tuples stored. The "IPv6 addresses" should be interpreted as the source addresses in the traffic from the Initiator to the DUT. In the IPv4 traffic, they are replaced by the public IPv4 address of the DUT. In the traffic from the Initiator to the DUT, the destination IPv6 addresses are actually *IPv4-embedded IPv6 addresses*, where the above-mentioned "IPv4 addresses" were appended to the 64:ff9b::/96 *NAT64 Well-Known Prefix* (WKP).

The results of the maximum connection establishment rate and throughput measurements are shown in Table XI and Table XII, respectively. It can be stated that the usage of multiple IP addresses caused no significant difference in the performance of the Jool stateful NAT64 implementation compared to the case where fix IP addresses were used. (The small performance decrease can be attributed to the higher number of elements in the ARP or NDP tables.)

C. *Stateful NAT64 Tests under OpenBSD using PF*

The measurement method described in [12] was reused with an important difference. Instead of deleting the connections with the **`pfctl -F states`** command, the DUT was rebooted after every single step of the binary search algorithm. It was done so to ensure a completely empty connection tracking table for each step because the above-mentioned command does not delete the complete content of the connection tracking table of PF, but it only "marks the states as expired, and then the purge scan is able to take them and actually free them" [19].

The same types of measurements using the same parameters

TABLE XI
MAXIMUM CONNECTION ESTABLISHMENT OF THE JOOL STATEFUL NAT64 IMPLEMENTATION, 4M CONNECTIONS

	In phase 1: pseudorandom enumeration of		
IP addresses	--	4,000*1,0000	10*10*
port numbers	40,000*100	--	*400*100
Median (fps)	577,879	542,059	559,947
Minimum (fps)	576,150	539,061	557,613
Maximum (fps)	578,614	543,504	562,531
Dispersion	0.43	0.82	0.88
Rel. perf. (%)	reference	93.80	96.90

TABLE XII
THROUGHPUT OF THE JOOL STATEFUL NAT64 IMPLEMENTATION, 4M CONNECTIONS, BIDIRECTIONAL TRAFFIC, PER DIRECTION RATES

	In phase 1: pseudorandom enumeration of		
IP addresses	--	4,000*1,0000	10*10*
port numbers	40,000*100	--	*400*100
Median (fps)	302,557	289,338	295,007
Minimum (fps)	301,170	289,015	294,332
Maximum (fps)	303,516	289,907	295,703
Dispersion	0.78	0.31	0.46
Rel. perf. (%)	reference	95.63	97.50

were executed as with Jool.

The results of the maximum connection establishment rate measurements are shown in Table XIII. The usage of pseudorandom IP addresses and fixed port numbers resulted in a slight (11.7%) increase of the maximum connection establishment rate compared to the case when fixed IP addresses and pseudorandom port numbers were used. When both the IP addresses and the port numbers were pseudorandom, the performance increase was only 7.24%. (The author assumes that the usage of 10 IP addresses on each side was probably not enough to achieve an even distribution of the interrupts on the CPU cores, but the investigation of this question is beyond the scope of the current paper.)

As far as the throughput tests are concerned, during the preliminary tests the author experienced that the steps of the binary search failed due to a very low number of missing frames in the reverse direction even at rather low frame rates. To handle this issue, a Loss Tolerance of 0.01% was used. It means that the given step of the binary search was considered "passed" if at least 99.99% of the frames arrived back to the Tester. (It was checked individually for each direction, and the condition had to be satisfied for both directions for passing the test.)

The results of the throughput measurements are shown in Table XIV. It is highly important that the usage of pseudorandom IP addresses resulted in significantly higher throughput then with fixed IP addresses. Unfortunately, the results show a rather high dispersion in all cases. For this reason, the author refrains from drawing conclusion from the fact that the combination of pseudorandom IP addresses and port numbers seem to result in a somewhat higher throughput than throughput of the case when only IP addresses were pseudorandom.

TABLE XIII
MAXIMUM CONNECTION ESTABLISHMENT OF THE PF STATEFUL NAT64 IMPLEMENTATION, 4M CONNECTIONS

	In phase 1: pseudorandom enumeration of		
IP addresses	--	4,000*1,0000	10*10*
port numbers	40,000*100	--	*400*100
Median (fps)	98,540	110,069	105,675
Minimum (fps)	97,532	108,791	104,701
Maximum (fps)	100,601	111,359	109,376
Dispersion	3.11	2.33	4.42
Rel. perf. (%)	reference	111.70	107.24

TABLE XIV
THROUGHPUT OF THE PF STATEFUL NAT64 IMPLEMENTATION, 4M CONNECTIONS, BIDIRECTIONAL TRAFFIC, PER DIRECTION RATES, BEWARE: LOSS TOLERANCE: 0.01%

	In phase 1: pseudorandom enumeration of		
IP addresses	--	4,000*1,0000	10*10*
port numbers	40,000*100	--	*400*100
Median (fps)	174,457	272,768	295,648
Minimum (fps)	129,279	238,616	246,676
Maximum (fps)	206,372	355,361	364,066
Dispersion	44.19	42.80	39.71
Rel. perf. (%)	reference	156.35	169.47

VIII. DISCUSSION AND FUTURE RESEARCH

OpenBSD 7.3 IPv4 packet forwarding throughput results in Table IX show that the usage of pseudorandom IP addresses caused a more than 3-fold performance increase compared to the cases when fixed IP addresses were used. This is a highly significant difference. As Internet traffic has multiple IP addresses, it means that in this case the RFC 2544 / RFC 4814 compliant laboratory test results did not reflect well the IPv4 packet forwarding performance of OpenBSD 7.3, as a result, they should be updated.

OpenBSD 7.3 IPv6 packet forwarding throughput results in Table X and OpenBSD 7.3 PF stateful NAT64 packet forwarding results in Table XIV also show more than 50 percent difference, which is also significant.

OpenBSD was used only as an example, several other various network interconnect devices may exist that do not support the setting of RSS, so that also the port numbers may be taken into consideration and thus their packet forwarding performance can show a rather significant difference when fixed IP addresses are used during laboratory testing and for forwarding Internet traffic. This methodological gap should be closed so that the results of the laboratory test may be more useful for both the manufactures and the users of network interconnect devices. To that end, the author discussed the issue with the chairs of the IETF BMWG and submitted the following Internet Draft [20] prior to the submission of the current paper for review.

The appropriate ranges for IP addresses to reflect the nature of the Internet traffic is beyond the scope of the current paper and it is considered an open question and an important topic for future research.

IX. Conclusion

It was pointed out that IETF BMWG documents lack guidelines for how to use pseudorandom IP addresses in stateless or stateful benchmarking. A solution was proposed to fill this methodological gap while honoring the constraints of the IPv4 and IPv6 address ranges reserved for benchmarking.

The **`siitperf`** free software stateless and stateful network performance tester program was extended to support the proposed solution. The performance penalty of the usage of pseudorandom IP addresses was measured and it was shown that the design goal of maintaining the high performance of **`siitperf`** was achieved.

The proposed solution was validated by performing both stateless and stateful benchmarking measurements. It was found that the proposed solution can give definitely different results than those produced using fixed IP addresses. With the help of the proposed method and the new version of **`siitperf`**, the laboratory benchmarking results of IPv4 and IPv6 routers, as well as those of stateful NAT64 gateways much better reflect the performance of the tested devices when they are used in production systems for forwarding Internet traffic.

Acknowledgements

The author thanks Fortix Consulting Ltd. for providing his research group with three Dell PowerEdge R730 servers.

The author thanks the National Media and Infocommunications Authority (NMHH) of Hungary for lending an Anritsu MP1590B Network Performance Tester.

The author thanks István Pilisi, NMHH, for his advice regarding the above-mentioned device and the Spirent SPT-N4U Tester, too.

The author thanks Keiichi Shima, Bertalan Kovács, István Pilisi, Szabolcs Szilágyi, and Ádám Bazsó for their reviewing and commenting on the manuscript.

The author thanks Vargáné Katalin Kiss and John Kowalchuk, Széchenyi István University, for the English language proofreading of the manuscript.

Appendix

A. Introduction to Siitperf

A brief introduction to **`siitperf`** is given to provide the reader with the essential information necessary to understand its extension to support multiple IP addresses. The following sections are based on author's open access papers [9], [11], [21], and [22] in which all the details can be found. Some of their text is reused.

1) The Stateless Version of Siitperf

The aim of the author was to design and implement a high performance and also flexible research tool. To that end, **`siitperf`** is a collection of binaries and shell scripts. The core measurements can be performed by one of three binaries, which are executed multiple times by one of four shell scripts. The binaries perform the sending and receiving of certain *Ethernet test frames containing IPv4 or IPv6 datagrams* (in short: *IPv4* or *IPv6 test frames*) at a pre-defined constant frame rate according to the test setup shown in Fig. 2. As **`siitperf`** allows X=Y, it can also be used for benchmarking an IPv4 or an IPv6 router. The shell scripts call the binaries supplying them with the proper command line parameters for the given core measurement.

The first two of the supported benchmarking procedures (*throughput* and *frame loss rate*) require only the above-mentioned sending of test frames at a constant rate and counting the received test frames, thus the core measurement of both procedures is the same. The difference is that the throughput measurement requires finding the highest rate at which the DUT can forward all test frames without loss, whereas the frame loss rate measurement requires performing the core measurement at various frame rates to determine the frame loss rate at those specific frame rates. The core measurement of both tests was implemented in the **`siitperf-tp`** binary and the two different benchmarking procedures were performed by two different bash shell scripts. The one used for determining the throughput uses a binary search to find the highest lossless frame rate with the predefined *error*, which expresses the stopping criterion for the binary search. It stops, when:

higher_limit – lower_limit <= *error*.

The core measurements of the latency and PDV benchmarking procedures were implemented by the **`siitperf-lat`** and **`siitperf-pdv`** binaries, respectively. They are different extensions of **`siitperf-tp`**.

Input parameters that are unchanged during the consecutive executions of the binaries are read from the **`siitperf.conf`** file, whereas those that are changed are supplied by the shell scripts as command line parameters.

The binaries were implemented in C++ using DPDK to achieve a high enough performance. An object oriented design was followed: the **`Throughput`** class served as a base class for the **`Latency`** and **`Pdv`** classes. The program structure of each C++ program is very simple: the main program reads the parameters first from the configuration file and then from the command line. Next, it calls the **`init()`** function of the required measurement, which initializes the *Environment Abstraction Layer* (EAL) of the DPDK, resets and starts the network interfaces, and performs a few sanity checks. Finally, the main program executes the proper measurement procedure. The measurement procedure prepares the parameters for the senders and receivers, and starts one sender and one receiver for each active direction (as separate threads). They are executed by their exclusively used CPU cores to ensure guaranteed performance. (The used CPU cores should be excluded from the scheduler of the Linux kernel using the **`isolcpus`** kernel command line parameter.) After the sender and receiver threads have finished, the main thread collects and evaluates their results. In a general case, when frame sending and receiving is active in both directions, two senders and two receivers are used, which are executed by

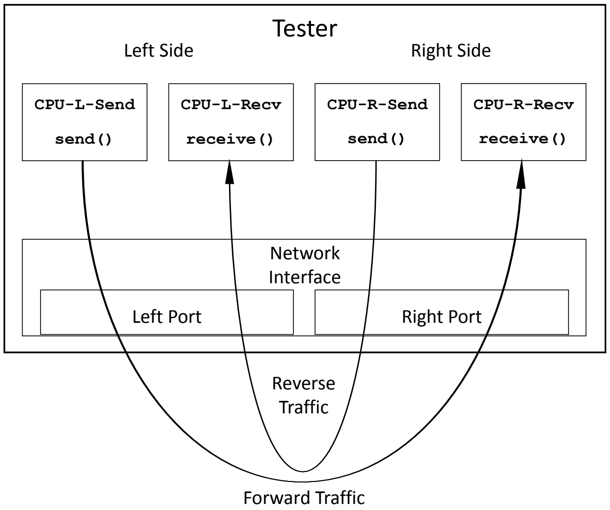

Fig. 11. Operation of the sender and receiver functions of **siitperf** during stateless testing. [9]

their respective CPU cores, as shown in Fig. 11. (Packets traversing through the DUT in the left to right direction are called *forward traffic* and the packets sent in the opposite direction are called *reverse traffic*.)

The **send()** and **receive()** functions are started by the **rte_eal_remote_launch()** function of DPDK, which does not allow the execution of non-static member functions. It was a serious limitation, thus the author could not carry out a fully object oriented design. The remotely executed functions can exchange data through a data structure, the address of which is provided to the functions as a pointer.

The first version of **siitperf** [4] used a high number of parameters in its configuration file to support flexibility. Its IP version could be set independently on its left and right sides using the two parameters that can be generated by the following brace expansion: **IP-{L,R}-Vers**. For each side (left and right) and both IP versions (4 and 6) the user could set two IP addresses: *real* and *virtual*. (The latter was used to represent an IP address from the other address family than that was actually used on the given side.) Table XV gives a short summary of how the eight potential IP addresses were used.

RFC 8219 also requires that besides the traffic that is translated (called "foreground traffic"), SIIT tests should also use non-translated native IPv6 traffic (called "background traffic"), and different proportions of the two types of traffic have to be used. Background traffic is normal IPv6 test frames and they are always sent from the "real" IPv6 address of the given side to the "real" IPv6 address of the other side. Background traffic is indistinguishable from the foreground test frames if the IP version of both sides is 6 (case no. 4).

As for the **receive()** function, it was written to be resilient. It does not take care of the IP version of the given side, but it checks the EtherType field of the frame to determine its IP version. It also checks if the received frame is a test frame. (To that end, **siitperf** writes the bytes of the "IDENTIFY" string as the first eight bytes of the UDP data field. It is not handled as a string, but as a 64-bit integer for performance considerations. [4])

Originally, **siitperf** literally followed the test frame format with fixed IP addresses and port numbers specified in the Appendix C.2.6.4 of RFC 2544. When the support for RFC 4814 pseudorandom port numbers was added [21], the flexible design of **siitperf** was kept; the user can specify the source and destination port number ranges for each direction separately and if the source and destination port numbers should have a fixed value, they should increase, decrease, or be pseudorandom. (Only the last one complies with RFC 4814.) These details are important regarding the design of the extension to support multiple IP addresses, as they should fit together. There are four parameters that describe the behavior of the port numbers. Their names can be obtained by the following brace expansion: **{Fwd,Rev}-var-{s,d}port**. The values of the parameters can be 0-3 with the following meanings: 0: fixed; 1: increasing; 2: decreasing; 3: pseudorandom. The ranges for the port numbers can be specified using 8 parameters: {**Fwd,Rev}-{s,d}port-{min,max}**. In all, there are 12 parameters used.

It is an important implementation detail that **siitperf** uses packet templates in which it modifies source and destination port numbers, as well as the appropriate 8-bit part of the IPv4 or IPv6 addresses, when multiple destination networks are used. IPv4 and UDP checksums are pre-calculated when the packet templates are generated (using 0 values for the fields to be modified) and they are modified according to the checksum of the modified fields. Depending on the IP version, pointers are set to the fields to be manipulated, and then the same code works for both IPv4 and IPv6 test frames.

Another important implementation detail was that only a single **send()** function was written and originally it had two sending loops: one for sending the same test frame using fixed IP addresses and port numbers, and another one for preparing several (up to 256) test frames the destination IP address of which belonged to different destination networks. When

Table XV. Specification of which parameters are used as source and destination IP addresses for foreground test frames on each side. (L/R means: Left/Right, the Virt(ual) value is used to represent an IP address from a different address family than used on the given side). [9]

Case	IP version		Type of the	IP addresses used by the Left Sender		IP addresses used by the Right Sender	
No.	Left	Right	DUT	source	destination	source	destination
1.	6	4	stateless NAT64 gw.	**IPv6-L-Real**	**IPv6-R-Virt**	**IPv4-R-Real**	**IPv4-L-Virt**
2.	4	6	stateless NAT46 gw.	**IPv4-L-Real**	**IPv4-R-Virt**	**IPv6-R-Real**	**IPv6-L-Virt**
3.	4	4	IPv4 router	**IPv4-L-Real**	**IPv4-R-Real**	**IPv4-R-Real**	**IPv4-L-Real**
4.	6	6	IPv6 router	**IPv6-L-Real**	**IPv6-R-Real**	**IPv6-R-Real**	**IPv6-L-Real**

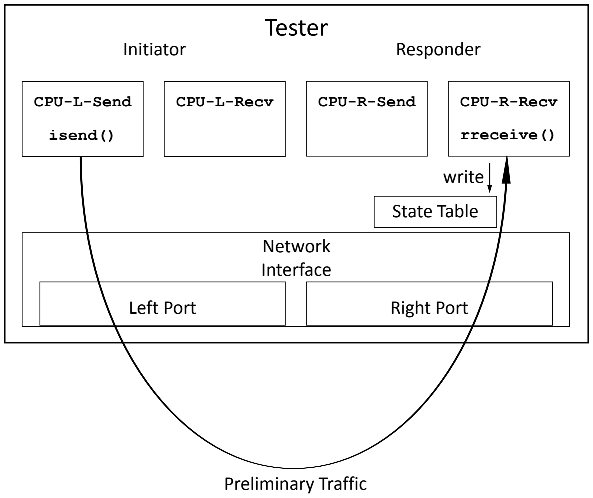

Fig. 12. Operation of the sender and receiver functions of **siitperf** during phase 1 of stateful testing. [9]

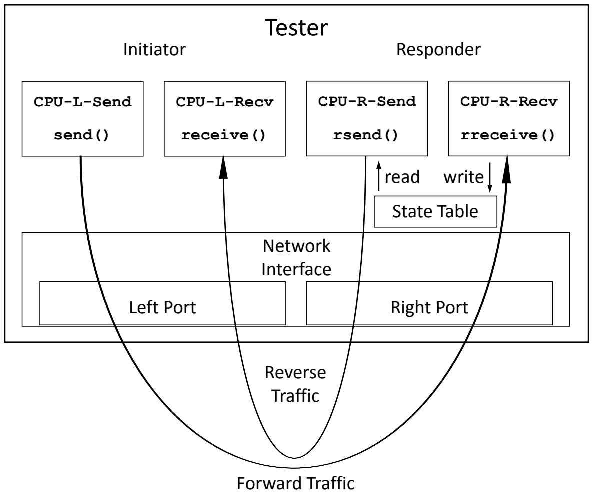

Fig. 13. Operation of the sender and receiver functions of **siitperf** during phase 2 of stateful testing. [9]

support for RFC 4814 pseudorandom port numbers was added, then the number of sending loops was doubled to support the original operation mode with fixed port numbers besides the new one with varying port numbers.

The following command line parameters are used for the throughput test:

- **IPv6 frames size** (in bytes), IPv4 frames are automatically 20 bytes shorter
- **frame rate** (in frames per second)
- **duration** of testing (in seconds)
- **global timeout** (in milliseconds), the tester stops receiving, when this global timeout elapsed after frame sending finished
- **n** and **m**: they are two relative prime numbers for specifying the proportion of foreground and background traffic: *m* packets form every *n* packets belong to the foreground traffic and the rest (*n-m*) packets belong to the background traffic.

Besides the parameters above, which are common for all the three binary programs, **siitperf-lat** and **siitperf-pdv** use various further ones, but they are not relevant to the current paper.

When the **send()** function finishes frame sending, it checks the duration of the frame sending. If it exceeds the desired duration by a factor higher than the predefined constant called "TOLERANCE" (the value of which is defined as 1.00001), it reports an error, and then bash shell script considers the test as failed. The aim of this checking is to avoid the kind of error situation that the test is performed at a longer time and thus at a lower frame rate then required due to the insufficient performance of the Tester.

2) Extension for Stateful Tests

The extension of **siitperf** for stateful NAT64 / NAT44 measurements is documented in [9].

The phase 1 operation of the Initiator is implemented by the new **isend()** function, which is able to provide the pseudorandom enumeration of all possible source port number and destination port number combinations required by the benchmarking methodology [10]. They are pre-generated before phase 1 using Durstenfeld's random shuffle algorithm [23]. Following the traditions of **siitperf**, the user has several factors of freedom; the port number enumeration is optional, and if it is used, increasing or decreasing order can also be used (besides pseudorandom), where the source port number is the low order counter and destination port number is the high order counter.

The operation of the sender and receiver functions of **siitperf** in stateful mode during phase 1 and phase 2 are shown in Fig. 12 and Fig. 13, respectively. In phase 1, the Initiator only sends packets using the **isend()** function, and it does not receive any packets. In phase 2, it sends and receives packets using the legacy **send()** and **receive()** functions.

It needs to be noted that the **isend()** function is much more general than required by the Internet Draft [10]. It is an extended version of the original **send()** function, keeping its all four packet sending loops and adding the optional functionality of port number enumeration. The only restriction is that port number enumeration may not be used together with multiple destination networks.

As for the Receiver, its implementation required two new functions: **rreceive()**and **rsend()**; and a new data structure: *state table*. The latter is implemented by an array of size *M* (specified by the user as command line parameter), the elements of which are *atomic* four tuples because it is concurrently read and written during phase 2. The **rreceive()** function extracts the source and destination IPv4 addresses and port numbers from the received IPv4 test frames and stores them in the state table. (The writing order is always increasing and its index is increased modulo *M*) The **rsend()** function prepares IPv4 test frames based on the four tuples taken from the state table (source and destination is swapped). The reading order can be increasing, decreasing and pseudorandom. (The latter is recommended.)

The stateful extension introduced only 3 new configuration file parameters. The name of the first one is **`Stateful`**, and its possible values and their meanings are: 0: perform stateless test; 1: perform stateful test, the Initiator is on the left side and the Responder is on the right side; 2: same as 1, but the Initiator and the Responder are on the opposite sides.

The second new parameter is **`Enumerate-ports`**, and its possible values and their meanings are: 0: no port number enumeration; 1 or 2: port numbers are enumerated in increasing or decreasing order; 3: port numbers are enumerated in pseudorandom order.

Notes regarding the values of the **`Enumerate-ports`** parameter:

- The value of 3 must be used to comply with the requirements of the Internet Draft [10]. The other values facilitate further opportunities for testing (e.g., to examine if the order of enumeration matters or not).
- Any non-zero value of the **`Enumerate-ports`** parameter overrides the values of the **`{Fwd,Rev}-var-{s,d}port`** parameters for phase 1.
- The zero value of the **`Enumerate-ports`** parameter results in the usage of the values of the **`Fwd-var-{s,d}port`** or **`Rev-var-{s,d}port`** parameters also in phase 1, depending on the 1 or 2 value of the **`Stateful`** parameter.

It is very important to note that port number enumeration applies only to the foreground traffic (traffic to be translated). The frames that belong to the background traffic (native IPv6 traffic) do not take part in the port number enumeration.

The third new parameter is **`Responder-ports`**, and its possible values and their meanings are: 0: a single fixed four tuple is used (like when a single test frame is always used); 1 or 2: the four tuples are taken from the state table in increasing or decreasing order; 3: the four tuples are selected from the state table in a pseudorandom way. Although the latter is recommended by the Internet Draft [10], reading the state table in increasing order provides a higher Tester performance due to less computation and caching [9].

The new command line parameters are to be interpreted as follows:

- **N**: the number of test frames to send in phase 1
- **M**: the number of entries in the state table of the Tester
- **R**: the frame rate, at which the test frames are sent during phase 1 (in frames per second)
- **T**: the global timeout for phase 1 frames (in milliseconds)
- **D**: the overall delay caused by phase 1 (in milliseconds)

It needs to be noted that phase 1 and phase 2 were originally called "preliminary phase" and "real test phase" [9]. This approach explains why those parameters were defined when **`siitperf`** supported only stateless tests, which were then applied to "the real test phase" (now referred to as phase 2), and when different parameters were needed, new ones were defined for the "preliminary phase" (now referred to as phase 1).

B. Validation of the Parameters

The parameter design is partially validated by setting the parameters to reflect the test setups mentioned in the previous sections of this paper.

Parameters for the traditional IPv4 routing tests with fixed IP addresses according to Fig. 1 and for the stateful NAT64 tests according to Fig. 3 are as follows:

```
IP-L-var 0 # fixed
IP-R-var 0 # fixed
```

Moreover, the values of the further new parameters are redundant and everything works as before.

For all the following test cases, they are to be set as follows (they are not repeated below):

```
IP-L-var 3 # pseudorandom
IP-R-var 3 # pseudorandom
```

Parameters for IPv4 router testing according to Fig. 4:

```
IP-L-min 2       # ".1" is for the DUT
IP-L-max 65534   # ".255.255" is broadcast
IP-R-min 2       # ".1" is for the DUT
IP-R-max 65534   # ".255.255" is broadcast
IPv4-L-offset 2 # last 16 bits
IPv4-R-offset 2 # last 16 bits
```

Parameters for IPv6 router testing according to Fig. 5:

```
IP-L-min 0        # The full range
IP-L-max 0xffff   # can be used.
IP-R-min 0        # The full range
IP-R-max 0xffff   # can be used.
IPv6-L-offset 12 # bits 96-111
IPv6-R-offset 12 # bits 96-111
```

Parameters for stateful NAT44 testing according to Fig. 6:

```
IP-L-min 0       # The full range
IP-L-max 65535   # can be used.
IP-R-min 0       # 0 is valid, but
IP-R-max 65534   # ".255.255" is broadcast.
IPv4-L-offset 1 # bits 8-23
IPv4-R-offset 2 # bits 16-31
```

Parameters for stateful NAT64 testing according to Fig. 7:

```
IP-L-min 0        # The full range
IP-L-max 0xffff   # can be used.
IP-R-min 0        # 0 is valid, but
IP-R-max 65534    # ".255.255" is broadcast.
IPv6-L-offset 12 # bits 96-111
IPv6-R-offset 14 # bits 112-127 (for IPv6-R-Virt!)
```

Parameters for stateful NAT44 testing according to Fig. 8:

```
IP-L-min 2       # ".1" is for the DUT
IP-L-max 65534   # ".255.255" is broadcast
IP-R-min 0       # 0 is valid, but
IP-R-max 65534   # ".255.255" is broadcast.
IPv4-L-offset 2 # bits 16-31
IPv4-R-offset 2 # bits 16-31
```

And for all stateful tests:

```
Enumerate-ips 3
```

Thus it was shown that the new parameters are suitable to express the settings required for the proposed test setups.

References

[1] S. Bradner, J. McQuaid, Benchmarking methodology for network interconnect devices, IETF RFC 2544, 1999. DOI: 10.17487/rfc2544

[2] T. Herbert, W. Brujin, "Scaling in the Linux networking stack", *Linux Kernel Documentation*, 2014, [Online], available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

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

[4] G. Lencse, “Design and implementation of a software tester for benchmarking stateless NAT64 gateways”, *IEICE Transactions on Communications*, vol. E104-B, no. 2, pp. 128-140. DOI: 10.1587/transcom.2019ebn0010

[5] C. Popoviciu, A. Hamza, D. Dugatkin, “IPv6 benchmarking methodology for network interconnect devices”, IETF RFC 5180, 2008. DOI: 10.17487/rfc5180

[6] M. Georgescu, L. Pislaru, G. Lencse, “Benchmarking methodology for IPv6 transition technologies”, IETF RFC 8219, 2017. DOI: 10.17487/rfc8219

[7] G. Lencse and Y. Kadobayashi, “Comprehensive survey of IPv6 transition technologies: A subjective classification for security analysis”, *IEICE Transactions on Communications*, vol. E102-B, no 10, pp. 2021–2035. DOI: 10.1587/transcom.2018ebr0002

[8] M. Bagnulo, P. Matthews, and I. Beijnum, “Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers”, *IETF RFC 6146*, Apr. 2011, DOI: 10.17487/RFC6146.

[9] 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, August 1, 2022, DOI: 10.1016/j.comcom.2022.05.028

[10] G. Lencse, K. Shima, “Benchmarking methodology for stateful NATxy gateways using RFC 4814 pseudorandom port numbers”, Internet Draft, Feb. 14, 2023, draft-ietf-bmwg-benchmarking-stateful-02 [Online], available: https://datatracker.ietf.org/doc/html/draft-ietf-bmwg-benchmarking-stateful-02

[11] G. Lencse and K. Shima, “Optimizing the performance of the iptables stateful NAT44 solution”, *Infocommunications Journal*, vol. 15, no. 1, March 2023, pp. 55-63, DOI: 10.36244/ICJ.2023.1.6

[12] G. Lencse, K. Shima, K. Cho, "Benchmarking methodology for stateful NAT64 gateways", *Computer Communications*, vol. 210, no. 1, October 1. 2023, pp. 256-272, DOI: 10.1016/j.comcom.2023.08.009

[13] G. Lencse, “Benchmarking Stateless NAT64 Implementations with a Standard Tester”, *Telecommunication Systems*, vol. 75, no. 3, pp. 245-257, 2020. DOI: 10.1007/s11235-020-00681-x

[14] C. Bao, X. Li, et al., IP/ICMP translation algorithm, IETF RFC 7915, DOI: 10.17487/rfc7915

[15] D. Scholz, “A look at Intel’s dataplane development kit”, *Proc. Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, Munich, 2014, pp. 115–122. DOI: 10.2313/NET-2014-08-1_15

[16] G. Lencse, Siitperf: An RFC 8219 compliant SIIT (stateless NAT64) tester, free software under GPLv3 license, [Online] Available: https://github.com/lencsegabor/siitperf

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

[18] S. Henderson, “Re: Does OpenBSD support Receive Side Scaling (also called: multi-queue receiving)”, 2022, the list archive of the OpenBSD MISC mailing list, [online], available: https://marc.info/?l=openbsd-misc&m=166581934723445&w=2.

[19] D. Gwynne, “Re: pf state-table-induced instability”, 2023, the list archive of the OpenBSD MISC mailing list, [online], available: https://marc.info/?l=openbsd-misc&m=169326012603726&w=2

[20] G. Lencse, K. Shima, “Recommendations for using multiple IP addresses in benchmarking tests”, Internet Draft, Oct. 20, 2023, draft-lencse-bmwg-multiple-ip-addresses-00 [Online], available: https://datatracker.ietf.org/doc/html/draft-lencse-bmwg-multiple-ip-addresses-00

[21] G. Lencse, “Adding RFC 4814 random port feature to siitperf: Design, implementation and performance estimation”, *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, vol. 9, no. 3, pp. 18–26. DOI: 10.11601/ijates.v9i3.291

[22] G. Lencse, “Checking the Accuracy of Siitperf”, *Infocommunications Journal*, vol. 13, no. 2, pp. 2-9, June 2021, DOI: 10.36244/ICJ.2021.2.1

[23] R. Durstenfeld, “Algorithm 235: Random permutation”, *Communications of the ACM*, vol. 7, no. 7, p. 420, July 1964, DOI: 10.1145/364520.364540

Gábor Lencse received his MSc and PhD in computer science from the Budapest University of Technology and Economics, Budapest, Hungary in 1994 and 2001, respectively.

He has been working full time for the Department of Telecommunications, Széchenyi István University, Győr, Hungary since 1997. Now, he is a Professor. He has been working part time for the Department of Networked Systems and Services, Budapest University of Technology and Economics, Budapest, Hungary as a Senior Research Fellow since 2005. The main area of his research is the performance and security analysis of IPv6 transition technologies. He is a co-author of RFC 8219.

Dr. Lencse is a member of the Institute of Electronics, Information and Communication Engineers (IEICE), Japan.