Design and Implementation of a Software Tester for Benchmarking Stateful NATxy Gateways: Theory and Practice of Extending Siitperf for Stateful Tests

Gábor Lencse[a,*]

[a]*Department of Telecommunications, Széchenyi István University, Egyetem tér 1, Győr, H-9026, Hungary*

Abstract

Our `siitperf` is the world's first RFC 8219 compliant free software SIIT (Stateless IP/ICMP Translation, also called stateless NAT64) benchmarking tool. It was written in C++ using DPDK (Intel Data Plane Development Kit). Our current effort aims to design and implement a test program for stateful NATxy gateways, including both stateful NAT64 and stateful NAT44 (also called NAPT: Network Address and Port Translation). Due to the object-oriented design of `siitperf`, it is feasible to extend it for stateful tests, while keeping its original design and features. In this paper, we introduce the problem of benchmarking stateful NATxy gateways and propose various solutions. We disclose the design and the most important implementation decisions of the stateful extension of `siitperf`. We prove the viability of our design and implementation by a functional NAT64 test and performing the maximum connection establishment rate, throughput, and frame loss rate measurements of the Jool stateful NAT64 implementation. We also carry out an initial performance estimation of the stateful extension of `siitperf`. Our tester is distributed as free software under the GPLv3 license for the benefit of the research, benchmarking and networking communities.

Keywords: benchmarking, IPv6 transition technology, performance analysis, stateful NAT44, stateful NAT64

1. Introduction

RFC 8219 [1] has defined a comprehensive benchmarking methodology for IPv6 transition technologies in 2017. To that end, it classified the high number of IPv6 transition technologies [2] into a small number of categories: dual stack, single translation, double translation, and encapsulation technologies. Both the SIIT [3] (Stateless IP/ICMP Translation, also called stateless NAT64) and the stateful NAT64 [4] IPv6 transition technologies belong to the single translation category.

We have created `siitperf` [5], the world's first RFC 8219 compliant free software SIIT benchmarking tool in 2019. We have implemented it in C++ using DPDK and documented its design, implementation, and initial performance estimation in [6]. As RFC 8219 reused the *throughput* benchmarking procedure from RFC 2544 [7], we have followed its *test frame format* using fixed source and destination UDP port numbers in our first implementation [6]. Then we have added the optional use of pseudorandom port numbers recommended by RFC 4814 [8] and documented the new feature in [9]. Our experience has shown that it was relatively easy and straightforward to extend `siitperf` to be able to use pseudorandom port numbers due to its object-oriented design, and we also managed to preserve its high performance [9].

Our current effort aims to extend `siitperf` to be able to benchmark stateful NAT64 gateways because they play an important role in the current phase of IPv6 transition [2]. However, in this paper, we point out that this extension is not at all straightforward, because of the missing theoretical background. We are not aware of any other working tester or publication, which would specify, how stateful NAT64, or even stateful NAT44 (also called NAPT: Network Address and Port Translation) gateways can be benchmarked using bidirectional traffic with random port numbers. Whereas our primary goal is the benchmarking of stateful NAT64 gateways, we consider the benchmarking of stateful NAT44 gateways also important and want to support it too. In theory, we design a method suitable for benchmarking any *stateful NATxy* gateway, where x and y are in $\{4, 6\}$.

The remainder of this paper is organized as follows. Section 2 contains a short survey of related work and then a general discussion on how stateful NATxy gateways may be benchmarked using bidirectional traffic with random port numbers. Section 3 gives a summary of the design and implementation of `siitperf` necessary to understand the following sections. Section 4 discloses our most important design considerations and implementation decisions. Section 5 summarizes the key points of our state-of-the-art benchmarking methodology for stateful NATxy gateways. Section 6 presents our functional tests and the maximum connection establishment rate, throughput, and frame loss rate measurements of the Jool [10] stateful NAT64 imple-

*Corresponding author
Email address: `lencse@sze.hu` (Gábor Lencse)

mentation, as well as an initial performance estimation of the stateful operation of `siitperf`. Section 7 provides a discussion and highlights our plans for further tests, development, performance optimization, and research on benchmarking methodology issues. Section 8 gives our conclusions.

2. Benchmarking Stateful NATxy Gateways using Bidirectional Traffic and Random Port Numbers

2.1. Related Work

In our short survey of relevant research results, we focus on the *performance analysis of stateful NAT64 gateways*. RFC 6146 [4] defined stateful NAT64 in 2011. During the following years, several papers have been published about the performance analysis of various stateful NAT64 solutions. Llanto and Yu [11] compared the performance of stateful NAT64 to that of stateful NAT44 through measuring RTT (Round-Trip Time) and "throughput". However, this "throughput" was measured using Apache Benchmark [12], and not an RFC 2544 compliant tester. Monte et al [13] compared the performance of stateful NAT64 to that of their own ALG (Application Layer Gateway) implementation. They also used Apache Benchmark to measure the connection time and the full access time of various websites. Yu and Carpenter [14] compared the performance of stateful NAT64 to that of the NAT-PT and an HTTP proxy. They used HTTP traffic with various request and response sizes, and they measured and compared the RTT of the mentioned three different solutions.

All these papers followed the approach that they measured the performance of a given NAT64 implementation along with a given DNS64 implementation. On the one hand, this could be ordinary (as stateful NAT64 is commonly used together with DNS64), however, the results reflect a kind of "weighted average" of the two and not the pure performance of the used NAT64 or DNS64 implementations. We have pointed out in [15] that: "even though both services are necessary for the complete operation, in a large network, they are usually provided by separate, independent devices; DNS64 is provided by a name server and NAT64 is performed by a router. Thus, the best implementation for the two services can be – and also should be – selected independently." To support this selection, we have compared the performance of four different DNS64 implementations under Linux, FreeBSD and OpenBSD [16] as well as we have compared the performance of the TAYGA [17] + `iptables` and OpenBSD PF stateful NAT64 implementations [15].

The common feature of all these measurements is that the traffic through the stateful NAT64 gateway happens in the following way:

1. First, a request is sent from the IPv6-only client to the IPv4-only server.
2. Then a reply is sent (or multiple replies are sent) from the IPv4-only server to the IPv6-only client.

On the one hand, this is ordinary, as connections through the stateful NAT64 gateway may be initiated only from the client-side. However, this measurement method is very far from the measurement method defined by the *de facto industry standard* RFC 2544 [7]. Its throughput measurement requires bidirectional traffic at a given constant frame rate. An elementary test lasts at least 60 seconds, while the Tester sends test frames through the DUT (Device Under Test) in both directions and counts the number of sent and received frames. If the number of received frames equals the number of sent frames, then the frame rate is increased and the test is re-run. Otherwise, the frame rate is decreased, and the test is re-run. (This is the official wording, but in practice, a *binary search* is used.) The *throughput* is the highest frame rate at which the number of received frames is equal to the number of sent frames.

In theory, RFC 2544 was IP version independent, but it was written with IPv4 in mind (e.g. IPv4 addresses are used in its examples). RFC 5180 [18] focused on IPv6, but it excluded IPv6 transition technologies from its scope. RFC 8219 addressed IPv6 transition technologies. It reused some measurement procedures from RFC 2544 (e.g. throughput, frame loss rate) redefined the latency measurement procedure, and added others (PDV and IPDV). Although RFC 8219 explicitly lists stateful NAT64 among the single translation technologies, but it says nothing about how the problem of the traffic in the IPv4 to IPv6 direction through the stateful NAT64 gateway is to be handled. In addition, RFC 4814 [8] requires the usage of a high number of different port number combinations in both directions. We have not found any publications resolving or at least discussing these challenges. Therefore, we do so in the following subsections.

2.2. Problem Formulation

As the problem is not specific to stateful NAT64, we discuss it in a general way. We use the example of the more well-known and widely used IPv4 *NAPT* (Network Address and Port Translation, please refer to Section 2.2 of RFC 3022 [19], it is also called *stateful NAT44*). NAPT is present in many places from small home networks to the largest ISP networks, where it is used in the CGN (Carrier-Grade NAT) gateway. Although we use IPv4 in our example to give an easy explanation of the problem, any IP version could be used. Fig. 1 shows the test and traffic setup for the throughput measurement of NAPT gateways. Although the arrows would suggest unidirectional traffic, RFC 8219 requires testing with bidirectional traffic, and testing with unidirectional traffic is optional. Following our naming convention used in [6] and [9], we call the direction following the arrows as *forward direction* and the opposite one as *reverse direction*. We used private IP addresses on the left side of the devices and public IP

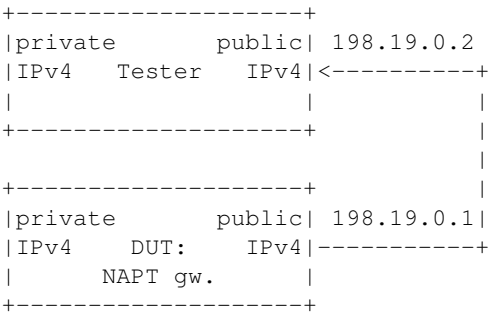

Figure 1: NAPT gateway test setup (based on RFC 2544)

addresses on their right side. Due to the operation of the NAPT solution, communication may only be initiated in the forward direction.

Now, we follow the possible operation of the test system. Let the left side port of the Tester send a *test frame* with the following IP addresses and port numbers: source: 10.0.0.2:10000, destination: 198.19.0.2:80, where the port numbers are arbitrary.

We note that the port numbers are *UDP port numbers*, because RFC 8219 requires testing with UDP traffic. We are aware that stateful translators use different timeout values for TCP and UDP "connections". Now, we follow the requirements of RFC 8219, but we return to this issue in section 7.

Let the *connection tracking table* of the NAPT gateway be empty at the beginning of testing, and let the NAPT gateway does not change the source port numbers when it is not necessary. Thus, the IP addresses and port numbers of the translated test frame are as follows: source: 198.19.0.1:10000, destination: 198.19.0.2:80. When the right-side port of the Tester receives the translated test frame, it may store the *four tuple* of IP addresses and port numbers, and then it can send a test frame with a *valid* four tuple that has a matching entry in the connection tracking table of the NAPT gateway. The identifiers of the test frame to be sent in the reverse direction are: source: 198.19.0.2:80, destination: 198.19.0.1:10000. The NAPT gateway translates back the test frame using the information of its connection tracking table, and the identifiers of the translated frame are: source: 198.19.0.2:80, destination: 10.0.0.2:10000.

Now, let us consider how pseudorandom source and destination port numbers can be used to comply with the requirements of RFC 4814. Their application in the reverse direction requires that preliminary traffic be provided in the forward direction *before* the actual throughput test: during this *preliminary phase*, the four tuples are observed and stored. After that, the right-side port of the Tester may randomly choose from among the stored four tuples to generate valid traffic that can be translated by the NAPT gateway.

Theoretically, pseudorandom source and destination port numbers could be used in the forward direction, however, this approach would be a *denial of service attack* against the NAPT gateway, because it would exhaust its connection tracking table. Let us see some calculations using the recommendations of RFC 4814:

- Recommended source port range: 1024-65535, its size is: 65535-1024+1=64512
- Recommended destination port range: 1-49151, its size is: 49151
- The number of source and destination port number combinations is: 64512*49151 = 3,170,829,312.

And yet we did not consider the requirement for testing with also 256 destination networks, which would further increase the number of connection tracking table entries.

Thus, we have shown that the Tester should not follow the recommendations of RFC 4814 for pseudorandom source and destination port numbers blindly. However, on the other hand, we agree with the purpose of RFC 4814, as we are aware that using the same fixed source and destination port numbers is very far from the operational conditions of NAPT gateways. Even a small home NAPT device has to handle a high number of different source port numbers since web browsers use a high number of concurrent TCP connections, the number of which depends on several factors including the content of the given web page, the type of client operating system and browser, etc., please refer to [20] for further details. A CGN NAPT gateway has to handle also a high number of different source IP addresses besides the high number of different source port numbers. These parameters have a significant influence on the number of connection tracking table entries and thus they should not be overlooked.

2.3. Possible Solutions

To find a reasonable solution, let us consider, what port numbers usually appear in the outgoing packets arriving at the NAPT gateway of an ISP. It is likely that:

- The source port numbers will be quite different in the range of 1024-65535.
- There will be a few very popular ones among the destination port numbers, with the dominance of 443 (HTTPS) and 80 (HTTP), appearing also the port numbers of several other widely used protocols[1] .

Theoretically, it could be possible to capture traffic at the NAPT gateway of an ISP, count the frequency of the occurrence of each source and destination port number, and store the statistics. One could implement a tester, which loads the statistics, and generates source and destination port numbers following the distributions recorded in the statistics. However, several different questions arise, for example:

[1]Please refer to the report of Internet Initiative Japan [21] for a particular observation of the popularity of the different protocols.

1. Are source and destination port numbers independent from each other or is there any correlation between them?
2. How much similar or different are the statistics of different NAPT gateways and how this difference influences the benchmarking results?
3. To what extent the statistics are permanent or changing over time, and how this possible change influences the benchmarking results?

The answer to the first question may simply make the random number generation a bit more complex, however, the answers to the second two questions may make it impossible to produce and publish meaningful benchmarking results that will be usable for others. We would like to build a more simple and easy-to-use model. Therefore, we make the following simplifications.

1. Let us omit the possible correlation of the source and destination port numbers.
2. Let us use uniform distribution for the source port numbers as recommended by RFC 4814. (Maybe its distribution is not uniform, but skewed, however, we hope that using uniform distribution is not a bad model.)
3. Let us also use uniform distribution for the destination port numbers, but in a much narrower range than it is recommended by RFC 4814. (This is a very significant simplification, which requires validation.)

The size of the destination port range can be used as a parameter and the performance of the NAPT gateway may be examined as a function of this parameter. The results may be useful when dimensioning a NAPT gateway.

3. Summary of `Siitperf`

In this section, we give a summary of the design and implementation of `siitperf` only to the extent necessary to understand the following sections. It is done by reusing some of the text of our open access papers [6] and [9], in which further details are available.

As for `siitperf`, we intended it to be a flexible tool designed for research and experimentation rather than an automated commodity Tester. Therefore, it is a combination of binaries and shell scripts. It supports the following benchmarking procedures: throughput, frame loss rate, latency, and PDV (packet delay variation). There are three binaries written in C++ using DPDK (Intel Data Plane Development Kit) [22] to ensure high enough performance. The binaries implement the core business logic and input a high number of parameters. There are four bash shell scripts (for the above-mentioned four benchmarking measurements), and they call the appropriate binary supplying the command line parameters necessary for the given measurement step. For example, the 20 repetitions and the binary search of the throughput test are performed by the `binary-rate-alg.sh` script, which calls the `siitperf-tp` binary for every 60 seconds long elementary test providing the required frame rate and several further parameters. The same `siitperf-tp` binary is used by the `frame-loss-rate.sh` script to measure the frame loss rate at various frame rates. Parameters that may vary among the consecutive executions of the binaries are supplied as command line parameters, whereas constant parameters (e.g. IP addresses, MAC addresses, etc.) are supplied in the `siitperf.conf` configuration file.

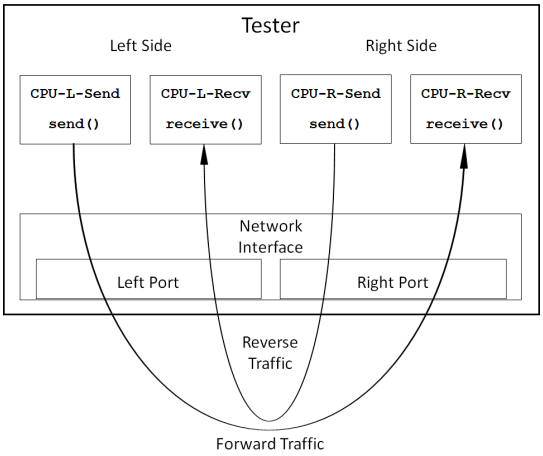

Figure 2: The operation of the sender and receiver functions of the original `siitperf`

We followed an object-oriented design. The classes for both the latency and the PDV measurements are extending their base class, throughput. (They are slightly different from each other, as the latency test uses only a specified number of timestamps, whereas the PDV test uses timestamps for every single frame.)

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 EAL (Environment Abstraction Layer) 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. After they have finished, the main thread collects and evaluates their results.

We show in Fig. 2, how the sender and receiver threads (that is the `send()` and `receive()` functions in the source code) are assigned to the CPU-s denoted with the self-explanatory names (`CPU-{L,R}-{Send,Recv}`) used in the configuration file.

From our point of view, it is important to mention that the four threads (two senders and two receivers) do not

Table 1: Specification of which parameters 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 the frame belongs to. Please refer to [6] for the details.)

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`

have any common data structures and they work independently from each other, except that:

- each receiver receives the test frames sent by the corresponding sender,
- receivers and senders on the same side use the same NIC (network interface card).

We have designed `siitperf` to be flexible due to using a high number of parameters. For example, the IP version can be specified individually and independently for each side, thus `siitperf` can also be used for testing IPv4 or IPv6 routers, not only SIIT gateways. When `siitperf` constructs and sends out test frames, their IP version always follows the IP version specified in the configuration file by the `IP-L-Vers` and the `IP-R-Vers` parameters for the Left Sender and the Right Sender, respectively. Table 1 summarizes which parameters are used as source and destination IP addresses for the test frames on each side.

RFC 8219 also requires that besides the traffic that is translated (we called it "foreground traffic"), tests should also use non-translated native IPv6 traffic (we called it "background traffic"), and different proportions of the two types of traffic have to be used. For us, it will be important that 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).

We note that a dual stack router may also be benchmarked using case no. 3 because besides the IPv4 foreground traffic, the background traffic is IPv6 and the proportion of the two may be set arbitrarily.

The proportion of the foreground traffic and background traffic can be expressed by two command line parameters called n and m, please refer to our original paper [6] for the details.

We note that the receiver function is resilient: it does not take care of the IP version of its side, it rather checks the value of the Type field of the Ethernet frame and processes the payload accordingly (as IPv4 or as IPv6). It does not check IP or MAC addresses, but it checks an 8-byte identifier to distinguish the test frames from other frames.

It is also important that RFC 2544 requires to use fixed source and destination IP addresses first, and then 256 destination networks for the benchmarking tests. We allow the user to specify the number of the networks on the left and right sides independently using any value from 1 to 256 in the configuration file:

```
Num-L-Nets 1 # Number of Left side networks
Num-R-Nets 1 # Number of Right side networks
```

The settings apply to both background and foreground traffic. But they are used only for destination networks and do not affect the source IP addresses.

There is a further parameter called `START_DELAY` (defined as a C preprocessor constant in the source file `defines.h`), which was originally intended to be typically technical: it facilitated the synchronized start of frame sending by the senders. (As their startup requires non-zero time, their frame sending has to be started at a well-defined time.) During our tests, frame loss was experienced at the beginning of the test, and it turned out that some part of the test system, perhaps the DUT (Device Under Test) was not yet ready, right after the initialization of the interfaces of the Tester. Thus, this parameter has received a new function to support a predefined delay between the starting of the network interfaces of the Tester and the starting of the actual measurement facilitating the proper initialization of the network interfaces of the DUT. Its default value was increased to 2 seconds and it may be further increased if needed.

Further parameters providing factors of freedom can be found in our original paper [6].

As for the extension of `siitperf` to use pseudorandom port numbers, we kept our flexible approach, and thus it can be specified individually for each direction and for the source and destination port numbers, whether they should be fixed or varying. If they are varying, they may be pseudorandom or increasing or decreasing in the consecutive frames. (The latter two are not RFC 4814 compliant, but they may be useful in some cases.) The configuration file allows to set the following parameters:

```
Fwd-var-sport 3
Fwd-var-dport 3
Rev-var-sport 1
Rev-var-dport 0
```

The numeric values are interpreted as follows:

0 fixed port number (the hard-wired value defined in Appendix C.2.6.4 of RFC 2544)

1 increasing port number,

2 decreasing port number

3 pseudorandom port number

It is computationally less expensive to use increasing (or decreasing) port numbers than using pseudorandom port numbers. Of course, not all combinations are useful, perhaps, there is no point in increasing both the source and the destination port numbers.

The configuration file shipped with `siitperf` contains the default settings for port number ranges as required by RFC 4814:

```
Fwd-sport-min 1024
Fwd-sport-max 65535
Fwd-dport-min 1
Fwd-dport-max 49151
Rev-sport-min 1024
Rev-sport-max 65535
Rev-dport-min 1
Rev-dport-max 49151
```

It is also an important implementation detail that the test frames are not built up from scratch during testing, but pre-generated test frames (templates) are modified to decrease the amount of work and, thus, to increase the maximum achievable frame rate.

We note that all sorts of variable port numbers apply to both foreground and background traffic.

As for the output of `siitperf-tp`, it reports the number of the transmitted frames and the received frames for the active directions (one direction may be missing):

```
Forward frames sent:
Forward frames received:
Reverse frames sent:
Reverse frames received:
```

It will be important that the bash shell scripts are expected to `grep` for the above expressions in the output of the program.

So far, we have mainly focused on the `siitperf-tp` throughput tester, which can also be used for the frame loss rate measurements. The design and the operation of the `siitperf-lat` latency tester are fairly similar. The main difference is that a certain number of frames are tagged for latency measurements. As the maximum number of latency frames is 50,000, they are always pregenerated. If the varying port number feature is used, then the port numbers are updated in the latency frames, too. When a tagged frame is sent, the sender function stores its timestamp and when a tagged frame is received, the receiver function stores its timestamp, too. After the latency test is finished, `siitperf-lat` processes the timestamps and calculates the typical latency and worst-case latency values for each active direction. The latency tester has two further command line parameters, the *delay* parameter specifies how much time after the start of the measurement the first tagged frame should be sent, and the *timestamps* parameter specifies the number of frames to be tagged.

The design and the operation of the `siitperf-pdv` PDV tester are even more straightforward extensions of `siitperf-tp`. It sends only *PDV test frames*, each of which contains an 8-byte ordinal number, which is used as an index for the array of the receiving and sending timestamps. These arrays are filled during the sending and receiving of the PDV test frames, and arrays are processed after finishing the measurement. The PDV tester has one further command line parameter called *frame timeout*. If the value of this parameter is 0, then the timestamp arrays are processed as required by RFC 8219 to calculate PDV. If the value of this parameter is higher than 0, then it is interpreted as the timeout parameter for each frame individually: those frames having higher latency than frame timeout are reclassified as *lost*. Hence, this implements a special throughput test, where the timeout is checked for each frame individually. Please refer to our original paper for the details and the justification of the method [6]. For us, this method is useful for determining the performance (maximum frame rate) of `siitperf-pdv`.

4. Design of the Stateful Extension of Siitperf

4.1. General Design Considerations

When we designed a functional extension of `siitperf`, we considered its compatibility with its previous versions very important. The new software should be able to perform all the original tests using the original parameters (in the command line and in the configuration file) and provide the original output. To do so, special values of the new parameters may be required, and if possible, these values should be their default values. (Thus, the usage of an old configuration file and command line parameters with the new software should result in its old way of operation.)

4.2. High-level Design Decisions

4.2.1. Considerations for Directions and Flexibility

Due to the nature of the stateful translation, it can only be used at most in one direction. To keep the flexibility of the software, we decided to let the user specify the direction. We also wanted to allow stateful translation to be combined with any IP version (4 or 6). From the set of possible combinations, stateful NAT44, stateful NAT64, and stateful NAT66 are surely meaningful. Stateful NAT46 [23] has also been proposed, but its Internet-Draft has never been published as an RFC.

4.2.2. Design of Stateful Testing

Regarding the stateful operation, let us name the roles of the two ports of the Tester as *Initiator* and *Responder*. As shown in Fig. 3, the Initiator resides on the "private"[2]

[2]We use IPv4 terminology to facilitate an easy understanding for those, who are more familiar with IPv4 than with IPv6. However, our design is not at all limited to stateful NAT44. Fig. 4 shows the test setup for benchmarking stateful NAT64 gateways.

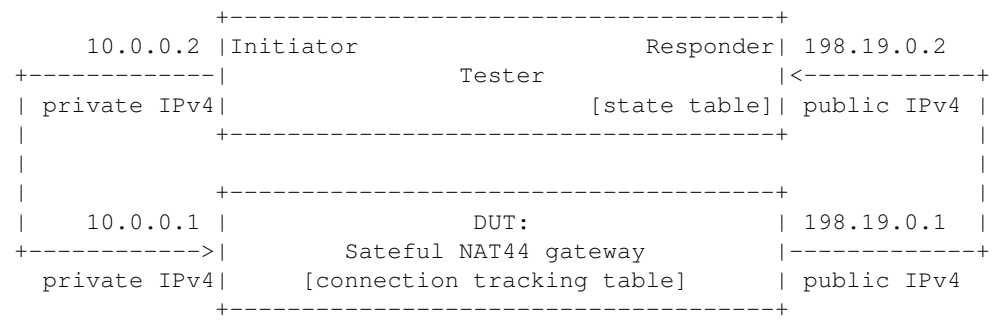

Figure 3: Test setup for benchmarking stateful NAT44 gateways

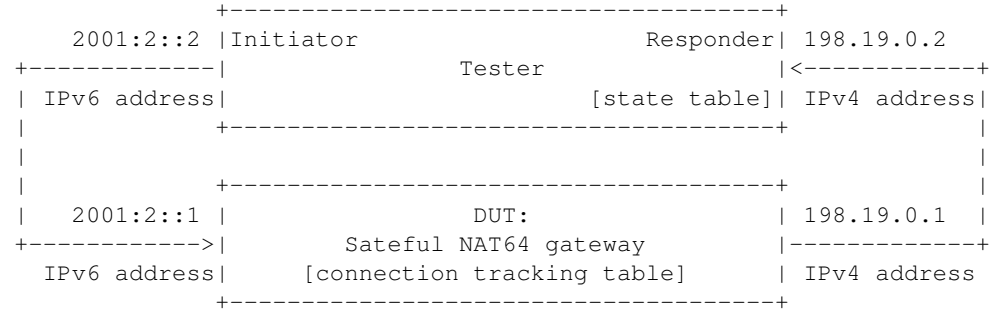

Figure 4: Test setup for benchmarking stateful NAT64 gateways

side of the DUT, and only the Initiator can initiate connection establishments due to the stateful nature of the DUT. The Responder resides on the "public" side of the DUT and it can send only test frames that belong to a connection already initiated by the Initiator. As both of them must be able to send proper test frames at the required frame rate from the very beginning of the test, a *preliminary phase* is necessary, while the Responder can observe and store enough *valid four tuples* (that belong to existing connections) in its *state table*. Thus, the Initiator and the Responder perform the following tasks:

- During the *preliminary phase*, the Initiator sends N number of test frames to the Responder through the DUT. The Responder extracts the IP addresses and the port numbers from the tests frames and stores them in its state table, but it does not send any test frames yet.
- During the *test phase*, the Initiator acts the same as the sender and receiver of the original `siitperf`. The Responder receives and processes the test frames as needed[3] and it further updates its state table on the basis of the IP address and port number information of the received frames. The responder also sends test frames using the IP addresses and port numbers from its state table.

[3]E.g. `siitperf-tp` simply counts them, whereas `siitperf-lat` and `siitperf-pdv` perform further tasks with timestamps.

As the Initiator is completely free to use any source and destination port number combinations during the test phase (even those not used during the preliminary phase), it is absolutely necessary for the Responder to update its state table during the test phase. This operation also means that *the sender and receiver of the Responder are no more independent*, but they have a common data structure, the state table, which is written by the receiver and read by the sender. Please refer to section 4.5 for the details.

4.3. Further Design and Implementation Decisions

4.3.1. Considerations for the State Table of the Responder

RFC 8219 defines black-box testing: the user is not aware of the internals of the DUT. In our case, it also means that we are not aware of even the size and policy of the connection tracking table of the DUT. We are *not able to keep the consistency* between the state table of the Responder and the connection tracking table of the DUT as we may not examine the latter. However, at least, we need to enable the user to control, how the old four tuples of IP addresses and port numbers are thrown out from the state table of the Responder. Allowing the user to specify a timeout could be handy from the user's perspective. However, its handling would consume a significant amount of processing power. Due to performance considerations, we decided to implement the state table of the Responder as a simple ring buffer of size M. If the test frames arrive at rate r, then the entries of the state table are overwrit-

ten in M/r time. (Please refer to section 4.3.6 for another consistency-related issue.)

4.3.2. Considerations for the Connection Establishment Rate

Usually, a high number of packets per connection are transmitted in a typical application scenario of stateful NATxy gateways. It also means that the connection establishment rate is significantly lower than the packet rate.

During the test phase of our benchmarking tests, the number of test frames per connection may be controlled by the number of possible four tuples (and also by M).

However, at the beginning of the preliminary phase, the initiator sends all different four tuples, that is, the connection establishment rate is equal to the frame rate. As the maximum connection establishment rate of a stateful device may be significantly lower than its maximum forwarding rate, we decided to enable the user to specify a different frame rate for the preliminary phase than the frame rate used in the test phase. Please see section 6.2, how `siitperf` supports the measurement of the maximum connection establishment rate of a stateful device.

4.3.3. Enumeration of Port Numbers

Our state-of-the-art benchmarking methodology for stateful NATxy gateways summarized in section 5, requires the pseudorandom enumeration of all possible port number combinations in the preliminary phase. In addition to that, we wanted to make `siitperf` also suitable for wilfully exhausting the port number range of a stateful NAT64 / NAT44 gateway for simulating a denial of service attack to support vulnerability analysis mentioned in [24] and [25].

Therefore, we have added a new input parameter *to combine source and destination port numbers into a single counter*. It means that the source port number is the lower two bytes and the destination port number is the higher two bytes of a 4-byte counter. However, its possible values are still limited by the specified ranges of the source and destination port numbers. (Please refer to section 4.3.5, how to set *port number enumeration*.)

We note that port number enumeration applies only to the translated traffic (called foreground traffic). The port numbers of the non-translated traffic (background traffic) do not take part in the enumeration.

We also note that port number enumeration is supported only in the preliminary phase.

4.3.4. Port Numbers of the Responder

Due to the stateful translation, the Responder has to generate test frames using the four tuples from its state table. It also means that *regarding foreground traffic*[4], the Responder should simply ignore various settings specified in the configuration file. (Namely: the number of destination networks and the port number ranges for the given direction as well as the values regarding the nature of the port numbers, that is, the 0, 1, 2 or 3 values of the `*-var-{d|s}port` parameters for the given direction.)

In order to keep resilience, now we consider, what approaches can be reasonable:

0 Use the fixed four tuple learned from the very first preliminary frame.

1 Take the next entry of the state table in increasing order.

2 Take the next entry of the state table in decreasing order.

3 Randomly select from among the state table entries.

We note that case 0 is the same approach, when hard-wired fixed port numbers are used in the original `siitperf`, literally following the test frame format in Appendix C.2.6.4 of RFC 2544.

We believe that case 3 is the true spirit of RFC 4814, whereas cases 1 and 2 are computationally less expensive alternatives. (At an early stage of the design of the benchmarking method there was a practical consideration that made at least one of them a must. We discuss it in section 4.3.6. However, later we found a better solution as described in section 5.)

4.3.5. New Input Parameters

Following our original policy that parameters that do not change during the execution of the shell scripts are put into the configuration file, we added the following parameters to the configuration file with the default value of 0:

```
Stateful 0 # valid values: 0, 1, 2
```

Its values have the following meanings:

0 The original operation of `siitperf` is kept, no new command line parameters are accepted.

1 Stateful test is performed, Initiator is on the left side and Responder is on the right side. New command line parameters are expected.

2 Stateful test is performed, Initiator is on the right side and Responder is on the left side. New command line parameters are expected.

We have introduced a configuration file parameter to control port number enumeration:

```
Enumerate-ports 0 # valid: 0, 1, 2, 3
```

Its values have the following meanings:

0 The original operation of `siitperf` is kept, the port numbers behave as usual.

[4] We note that the original settings still apply for the background traffic.

1 The port numbers are enumerated in *increasing* order (source port number is the low order counter and destination port number is the high order counter), but the source and destination port numbers are limited to their specified ranges.

2 Like "1", but the order of enumeration is *decreasing*.

3 All possible combinations of the available port numbers specified by the source and destination port number ranges are enumerated in a *pseudorandom order*.

We note that port number enumeration applies only for the foreground traffic, and it is available only when a single destination network is set, otherwise, the program gives an "Input Error:" message.

To express the policy, how the consecutive four tuples are selected from the state table of the Responder for the foreground traffic, we introduced the following configuration file parameter:

```
Responder-ports 0 # valid: 0, 1, 2, 3
```

The interpretation is defined by the listed items in section 4.3.4.

As for the new command line parameters, they follow the command line parameters of the throughput test, and they precede the additional parameters of the Latency and PDV measurements.

They are to be specified in the following order:

N $(1 - 2^{32} - 1)$ – the number of test frames to send in the preliminary phase

M $(1 - 2^{32} - 1)$ – the number of entries in the state table of the Tester

R (in frames per second) – the frame rate, at which the test frames are sent during the preliminary phase

T (in milliseconds, 1 – 2,000) – the global timeout for the preliminary frames

D (in milliseconds, 1 – 100,000,000) – the overall delay caused by the preliminary phase

We note that N denotes the number of *all* frames (including foreground and background frames) sent during the preliminary phase.

It is important that the sending of the N number of test frames at the specified R frame rate should happen and also the T global timeout should elapse within the D time, otherwise `siitperf` reports an error message and exits.

We note that setting M to 1 is allowed only in the case if `Responder-ports` is set to 0. Please refer to section 4.3.8 for an explanation.

4.3.6. The Issue of Active Directions

So far, we considered the general case, when both directions are active, that is, bidirectional traffic is used for benchmarking. As it is in stateless testing, any of the two directions may be set inactive also in the case of stateful testing. It is trivially not a problem if traffic flows only from the Initiator to the Responder. When traffic flows only from the Responder to the Initiator, then the state table of the Responder is filled during the preliminary phase and it remains unchanged during the testing phase. It may cause a serious problem under certain conditions. Stateful NAT64 or NAT44 gateways use various timeout values for the connections. Let us consider the following situation. If traffic flows only from the Responder to the Initiator during the test phase, and the Responder uses pseudorandom four tuple selection, it may happen that a specific four tuple is not used for a specific timeout and then it is used again. It results in the construction of a frame that belongs to a no more existing connection in the gateway. Therefore, it is dropped by the gateway, and the loss of the frame causes the throughput test to fail. This issue is properly solved by using an appropriate timeout, please refer to section 5 for the details.

4.3.7. The Issue of Indistinguishable IPv6 Background Frames

When the IP version is 4 on the side where the Responder resides, then frames translated by either stateful NAT44 or stateful NAT64 arrive as IPv4 frames, and IPv6 frames belong to the background traffic. Hence, foreground and background frames can be easily distinguished by the IP version. However, when the IP version is 6 on the side where the Responder resides, then frames translated by either stateful NAT46 or stateful NAT66 arrive as IPv6 frames, and they are indistinguishable from the background traffic using only the IP version. The problem could be easily solved by using a different 8-byte identifier for the test frames belonging to the background traffic or by examining also the source IPv6 address. However, we did not implement it yet, please refer to section 4.4.1 for more details.

4.3.8. The Issue of Inter-thread Communication

Both high performance and flexibility were our primary design concerns. As inter-thread communication may negatively influence performance, we had to make a compromise on the following issue.

Originally, we planned to allow the partial filling of the state table of the Tester during the preliminary phase, and the receiver of the Responder could fill the remaining entries in the test phase. However, it would have required continuous communication of the number of valid entries from the receiver of the Responder to the sender of the Responder, which could have a significant impact on the performance of the Tester. Although it could have been stopped after filling the state table, it would further complicate the code, whereas a single extra "if" statement in

the innermost receiving and sending loops was also considered a hindrance to be avoided. So, we decided that the state table must be filled in the preliminary phase.

Writing and reading of the state table may slow down the Tester only in the case if the same entry is affected. Therefore, we decided to support fixed port numbers by a separate code, which does not continuously write and read the single entry. In this case, the very first entry of the state table is read *only once* at the beginning of the test phase, and then the sender and the receiver work independently.

4.4. Implementation of the Stateful Tests

4.4.1. Scope Decisions

Considering our limited time and the vast difference between the deployment of stateful NAT44 and stateful NAT64 versus stateful NAT46 and stateful NAT66, we decided to support only the first two of them. (The support for the latter two is not an intellectual challenge, but requires a significant amount of coding and testing.)

Our decision means that the Initiator has to be able to handle both IPv4 and IPv6, but the Responder needs to be able to handle only IPv4 as foreground traffic.

4.4.2. Design of the Initiator

As we mentioned before, the sender of the Initiator is a modified version of the sender function of the stateless `siitperf`. The main difference is the support for port number enumeration using a twice two-byte counter in the preliminary phase[5]. Let us see an example. If the source port numbers are set to increase from 10,000 to 49,999 (40,000 different values) and the destination port numbers are set to increase from 80 to 179 (100 different values) then 40,000*100 = 4,000,000 different combinations can be enumerated.

- If the sender of the Initiator has to enumerate the available port number combinations in a pseudorandom order, then it is checked, if there are enough unique port number combinations, and if not, then an `Error` is reported. (It is so to support proper measurements as described in Section 5.)
- If increasing or decreasing port number enumeration is required, then no such check is performed, and the counter of the combined source and destination port numbers is allowed to wrap around. (It is so not to limit the usability of `siitperf` as a denial of service attack testing tool.)

Port number enumeration is supported only in the case when the number of destination networks is set to 1.

During the operation of `siitperf`, frame sending and receiving happens twice: first, in the preliminary phase, and second, in the test phase. To protect the bash shell scripts processing the output of `siitperf` from confusion, `siitperf` uses the word "Preliminary" instead of "Forward" or "Reverse", when reporting the number of frames sent and received in the preliminary phase.

As for the receiver function, it is not used on the Initiator side during the preliminary phase, and the original one was kept in the test phase.

4.4.3. Design of the Receiver of the Responder

The consistency of the state table entries is ensured using atomic variables of C++. The type of the entries of the state table is defined as follows:

```
typedef std::atomic<fourTuple> atomicFourTuple;
```

Hence, both the reading and the writing of the entries of the state table are atomic operations.

The receiver of the Responder extracts the IPv4 addresses and port numbers from the received IPv4 test frames and writes them first into a local variable of type `struct fourTuple`, then it writes the four tuples into the state table in increasing order starting from index 0.

We note that neither the receiver nor the sender of the Responder converts IP addresses and port numbers between network byte order and host byte order because they are only copied but not manipulated.

4.4.4. Design of the Sender of the Responder

The sender of the Responder supports multiple modes of operation. If `Responder-ports` is set to 0, then a single IPv4 test frame is generated based on the very first element of the state table (index 0), and always this frame is sent as foreground traffic without regard to the number of destination networks. Background traffic is generated using fixed port numbers, but multiple destination networks may be used.

If `Responder-ports` is set to 1, 2, or 3, then all the entries of the state table are used as specified in section 4.3.4.

Following our original approach, we used pre-generated templates of Test Frames and modified their IP addresses and port numbers.

4.4.5. Design of the Latency Measurements

So far, we focused on the design of the stateful extension of the `siitperf-tp` throughput tester. The extension of the `siitperf-lat` latency tester is fairly similar, most things are quite straightforward. We mention only a few differences. As no tagged frames are sent during the preliminary phase, the Initiator of the throughput tester and the receiver of the Responder of the throughput tester are reused in the preliminary phase.

As with the throughput tests, port number enumeration is supported only in the preliminary phase of the latency measurements. (The program gives a warning about it if port number enumeration is specified in the configuration file.)

[5]Port number enumeration is supported only in the preliminary phase. In the test phase, the stateless sender is reused as the sender of the Initiator.

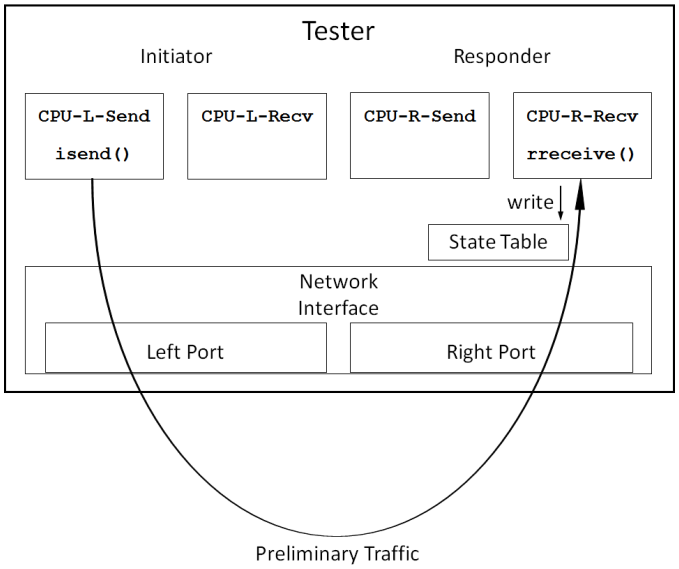

Figure 5: The operation of sender and receiver functions of the stateful `siitperf` during the preliminary phase

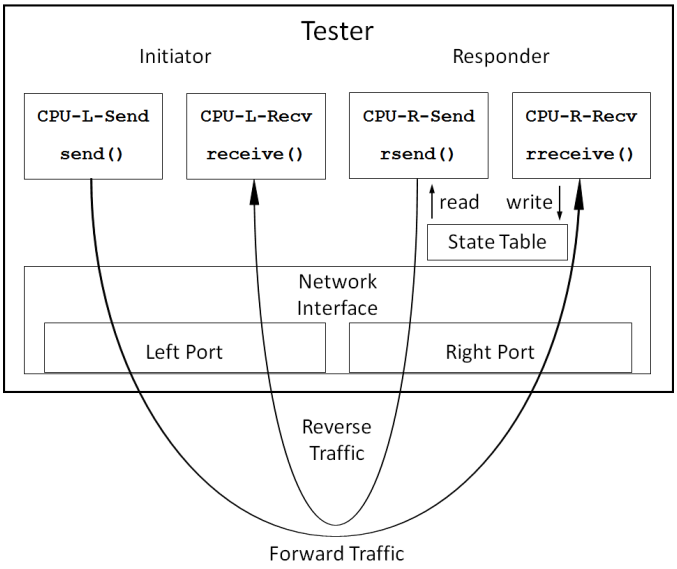

Figure 6: The operation of sender and receiver functions of the stateful `siitperf` during the test phase (using bidirectional traffic)

We note that *latency frames* (test frames tagged for latency measurements) are pre-generated and used as templates: they are modified in the same way as the templates of the normal test frames, the only difference is that they are used only once.

4.4.6. Design of the PDV Measurements

The extension of the `siitperf-pdv` PDV tester was completely straightforward. We followed the same approach as with the latency tester: the Initiator of the throughput tester and the receiver of the Responder of the throughput tester are reused in the preliminary phase and port number enumeration is not supported in the test phase.

4.4.7. Implementation of the Pseudorandom Enumeration of the Port Numbers

As the pseudorandom enumeration of all the available port number combinations is very important for our state-of-the-art measurement method described in section 5, we disclose its implementation details.

The pseudorandom port number pairs are generated *before* the beginning of the preliminary phase by the CPU core which is later used for the execution of the sender of the Initiator to ensure the allocation of NUMA local memory for the array of the pre-generated port numbers. First, all possible port number combinations (determined by the source and destination port number ranges) are enumerated in the array of port number combinations in increasing order, and then they are put into pseudorandom order using Dustenfeld's random shuffle algorithm [26].

4.5. Summary of the Sending and Receiving Functions

Now we summarize, what was changed and what was kept from the sending and receiving functions of the original `siitperf`, as well as when they operate during a complete throughput test.

We suppose that the value of the `Stateful` parameter is set to 1, that is, the Initiator is on the left side and the Responder is on the right side.

During the *preliminary phase*, the Sender function of the Initiator (called `isend()`) sends preliminary frames, and the receiver function of the Responder (called `rreceive()`) receives them, and extracts and stores the four tuples into its state table, as shown in Fig. 5.

During the *test phase*, the Initiator acts completely the same as in the stateless version. The Responder uses its new `rreceive()` and `rsend()` functions to receive and send frames. They are not independent from each other, because they are interconnected by the state table, written by the receiver and read by the sender of the Responder, as shown in Fig. 6.

5. State-of-the-Art Benchmarking Method

Until we published it as an Internet-Draft [27], there was no systematic proposal for benchmarking stateful NATxy gateways. The basic idea of the measurement method is to ensure that:

1. During the *preliminary phase*, all test frames result in the establishment of a new connection in the DUT.
2. During the *test phase*, no new connections are established in the connection tracking table of the DUT.
3. The connection tracking table of the DUT is empty at the beginning of the preliminary phase, and no connections are deleted from there until the end of the test phase.

These conditions are necessary so that the maximum connection establishment rate measurement (performed in

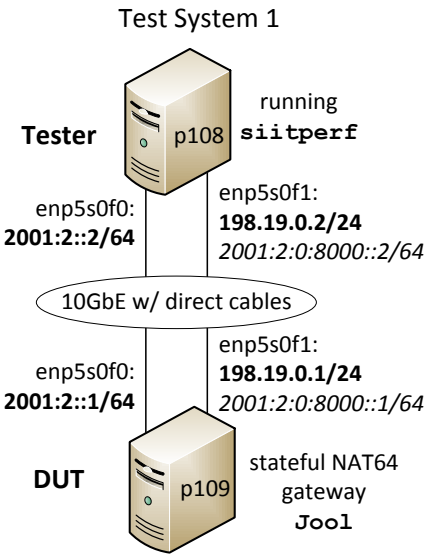

Figure 7: Test system for stateful NAT64 tests with Jool

the preliminary phase) and all other measurements (e.g. throughput, latency, etc.) performed in the test phase give clear and repeatable results. To that end, it is necessary to:

1. Use all different and pseudorandom port number combinations for all test frames during the preliminary phase.
2. Enumerate all possible port number combinations (determined by the specified source and destination port number ranges) in the preliminary phase.
3. Set the timeout in the DUT to a higher value than the length of the entire experiment.
4. Make sure that the capacity of the connection tracking table of the DUT is large enough to store all the connections (defined by the number of all possible port number combinations).
5. Start each experiment with an empty connection tracking table of the DUT.

This method proved to be viable when we used it for measuring the scalability of the `iptables` stateful NAT44 implementation up to 800 million connections and that of the Jool [10] stateful NAT64 implementation up to 1.6 billion connections [28].

6. Functional and Performance Tests

The aim of this section is threefold:

1. to demonstrate the operation of the stateful NAT64 measurements,
2. to test the usability of our Tester in a typical application scenario,

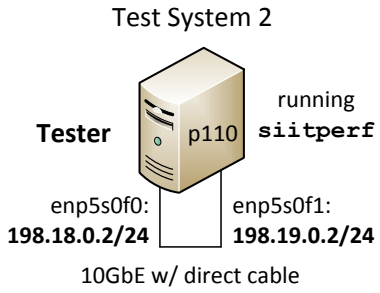

Figure 8: Test system for determining the performance limits of the stateful operation of `siitperf`

3. to make an initial performance assessment of the stateful operation of `siitperf`.

As a test environment, we used three "P" series nodes (p108, p109, p110) of NICT StarBED[6], Japan. They are Dell PowerEdge R430 servers with two 2.1GHz Intel Xeon E5-2683 v4 CPUs having 16 cores each, 384GB 2400MHz DDR4 SDRAM, and Intel 10G dual-port X540 network adapters. Hyper-threading was switched off and the clock frequency of all servers was set to 2.1GHz (fixed) using the `tlp` Linux package.

We used two test setups with different goals. The aim of *Test System 1* (shown in Fig. 7) was to demonstrate the operation of a stateful NAT64 measurement and to perform the most important benchmarking measurements of the Jool [10] stateful NAT64 implementation. *Test System 2* (Fig. 8) was used to perform an initial performance estimation of `siitperf`.

The Debian Linux 9.13 operating system was used on p108 and p110 computers. The Linux kernel version was: 4.9.0-4-amd64. The DPDK version was 16.11.11-1+deb9u2. The Debian Linux operating system was updated to version 11.2 on p109 because that version contains Jool in its package set. The Linux kernel version was: 5.10.0-11-amd64. The Jool version was 4.1.5-1.

6.1. Demonstration of a Stateful NAT64 Test

We have tested the functional operation of the stateful NAT64 measurement using *Test System 1*, the topology of which is shown in Fig. 7. The Tester and the DUT were interconnected by two 10GbE direct cable links. IPv6 was used on the left side network interfaces of the devices, and IPv4 was used on their right side. (IPv6 addresses are also assigned to the right side interfaces to facilitate "background" traffic, which is native IPv6 and not translated.) Stateful NAT64 was implemented by Jool [10]. We used the 64:ff9b::/64 NAT64 well-known prefix to construct the IPv4-embedded IPv4 address as follows: 64:ff9b::198.19.0.2.

Jool was set up by the following commands:

[6] http://starbed.nict.go.jp/en/aboutus/index.html

```
 1 0.000000000 fe80::a236:9fff:fec5:d2e4 --> ff02::16     ICMPv6 170 Multicast Listener Report Message v2
 2 0.459988669 fe80::a236:9fff:fec5:d2e4 --> ff02::16     ICMPv6 170 Multicast Listener Report Message v2
 3 2.097980439    2001:2::2 --> 64:ff9b::c613:2 UDP 80 10000 --> 80 Len=18
 4 2.297974826    2001:2::2 --> 64:ff9b::c613:2 UDP 80 10001 --> 80 Len=18
 5 2.497975000    2001:2::2 --> 64:ff9b::c613:2 UDP 80 10002 --> 80 Len=18
 6 2.697973421    2001:2::2 --> 64:ff9b::c613:2 UDP 80 10003 --> 80 Len=18
 7 2.897973228    2001:2::2 --> 2001:2:0:8000::2 UDP 80 28350 --> 101 Len=18
 8 4.097972108    2001:2::2 --> 64:ff9b::c613:2 UDP 80 48819 --> 81 Len=18
 9 4.098016609 64:ff9b::c613:2 --> 2001:2::2    UDP 80 80 --> 10003 Len=18
10 4.297971298    2001:2::2 --> 64:ff9b::c613:2 UDP 80 46256 --> 97 Len=18
11 4.297993205 64:ff9b::c613:2 --> 2001:2::2    UDP 80 80 --> 10002 Len=18
12 4.497969803    2001:2::2 --> 64:ff9b::c613:2 UDP 80 38671 --> 129 Len=18
13 4.497973784 64:ff9b::c613:2 --> 2001:2::2    UDP 80 80 --> 10003 Len=18
14 4.697971095    2001:2::2 --> 64:ff9b::c613:2 UDP 80 38875 --> 161 Len=18
15 4.697975518 64:ff9b::c613:2 --> 2001:2::2    UDP 80 80 --> 10002 Len=18
16 4.897968973    2001:2::2 --> 2001:2:0:8000::2 UDP 80 33935 --> 142 Len=18
17 4.897977901 2001:2:0:8000::2 --> 2001:2::2    UDP 80 29900 --> 23331 Len=18
```

Figure 9: The `tshark` capture of a stateful NAT64 test on the `enp5s0f0` interface of the DUT

```
modprobe jool
jool instance add --netfilter \
  --pool6 64:ff9b::/96
jool pool4 add 198.19.0.1 --udp 1-65535
```

To demonstrate the operation of the stateful NAT64 test, we performed a very short and low rate test. Only five preliminary frames were sent: 4 foreground frames and 1 background frame (to demonstrate it too). We used port number enumeration, and the Responder selected the four tuples randomly.

The new configuration file parameters were set as follows:

```
Stateful 1 # yes, Initiator is on the Left
Enumerate-ports 1 # yes, in increasing order
Responder-ports 3 # 4-tuples random select
```

We used port number enumeration in increasing order instead pseudorandom enumeration to facilitate an easy observation.

The command line was:

```
siitperf-tp 84 5 1 2000 5 4 5 4 5 500 2000
```

The first 6 command line parameters were "inherited" from the command line of the stateless tester. They denote that:

- The IPv6 frame size was 84 bytes (64 bytes for IPv4).
- The frame rate was 5 frames/s (in each direction).
- The test duration was 1 second.
- The global timeout was 2000ms.
- The value of n was 5 and the value of m was 4: it means that 4 of every 5 frames belonged to the foreground traffic.

The next 5 parameters are new:

- $N = 5$ preliminary frames were sent by the Initiator.
- The size of the state table of the Responder was $M = 4$.
- The preliminary frame rate was $R = 5$ frames/s.
- The global timeout for the preliminary phase was $T =$ 500ms.
- The total delay caused by the preliminary phase was $D = 2000$ms. (It includes the sending of the preliminary frames, the global timeout of the preliminary phase and the waiting time before the real test phase.)

We have captured the traffic by `tshark` on both network interfaces of the DUT: `enp5s0f0` and `enp5s0f1`, and they are shown in Fig. 9 and Fig. 10, respectively. As `siitperf` resets the network interfaces, the first two lines of both figures contain IPv6 multicast messages. (As `tshark` starts the time measurement from the arrival of the first frame, the times of the two captures are synchronized approximately, but not completely.)

In both figures, frames 3-6 are the foreground preliminary frames. In Fig. 9, the 64:ff9b::c613:2 IPv6 destination address represents the 198.19.0.2 IPv4 address shown in Fig. 10 as the destination address. And the 2001:2::2 source IPv6 address was replaced with 198.19.0.1 by Jool. Port number enumeration in increasing order can also be observed: the source port numbers start from 10,000 and increase by 1 on the IPv6 side. Jool maps the consecutive source port numbers to different, but also consecutive source port numbers, and currently it happens from 4,127.

As frame 7 is a background frame (native IPv6), the stateful NAT64 gateway leaves it unchanged. Its port numbers are pseudorandom, as background frames do not take part in the port number enumeration.

Frames 8-17 were sent during the test phase. Now the port numbers of the "forward" direction frames are random. The port numbers of the 4 foreground frames in the "reverse" direction frames are determined by the pseudorandom selection of the four tuples.

```
 1 0.000000000 fe80::a236:9fff:fec5:d2e6 --> ff02::16     ICMPv6 170 Multicast Listener Report Message v2
 2 0.519996603 fe80::a236:9fff:fec5:d2e6 --> ff02::16     ICMPv6 170 Multicast Listener Report Message v2
 3 1.938026102   198.19.0.1 --> 198.19.0.2   UDP 60 4127 --> 80 Len=18
 4 2.138011732   198.19.0.1 --> 198.19.0.2   UDP 60 4128 --> 80 Len=18
 5 2.338011860   198.19.0.1 --> 198.19.0.2   UDP 60 4129 --> 80 Len=18
 6 2.537998467   198.19.0.1 --> 198.19.0.2   UDP 60 4130 --> 80 Len=18
 7 2.738005376    2001:2::2 --> 2001:2:0:8000::2 UDP 80 28350 --> 101 Len=18
 8 3.938008055   198.19.0.1 --> 198.19.0.2   UDP 60 61309 --> 81 Len=18
 9 3.938013475   198.19.0.2 --> 198.19.0.1   UDP 60 80 --> 4130 Len=18
10 4.137989195   198.19.0.1 --> 198.19.0.2   UDP 60 9342 --> 97 Len=18
11 4.137993463   198.19.0.2 --> 198.19.0.1   UDP 60 80 --> 4129 Len=18
12 4.337977383   198.19.0.2 --> 198.19.0.1   UDP 60 80 --> 4130 Len=18
13 4.337994824   198.19.0.1 --> 198.19.0.2   UDP 60 21452 --> 129 Len=18
14 4.537976874   198.19.0.2 --> 198.19.0.1   UDP 60 80 --> 4129 Len=18
15 4.538007685   198.19.0.1 --> 198.19.0.2   UDP 60 60800 --> 161 Len=18
16 4.737977770 2001:2:0:8000::2 --> 2001:2::2    UDP 80 29900 --> 23331 Len=18
17 4.737985520    2001:2::2 --> 2001:2:0:8000::2 UDP 80 33935 --> 142 Len=18
```

Figure 10: The `tshark` capture of a stateful NAT64 test on the `enp5s0f0` interface of the DUT

We note that we used only a single public IPv4 address on the IPv4 interface of the stateful NAT64 gateway, but using multiple public IPv4 addresses could cause no problem, as the Responder stores the entire four tuples and uses their elements for traffic generation.

6.2. Maximum Connection Establishment Rate Measurement

Before an actual stateful NAT64 throughput test could be performed, one must determine the maximum connection establishment rate, and a rate somewhat lower than that should be used during the preliminary phase of the throughput test to prevent the failure of the measurement during the preliminary phase due to frame loss caused by an improper frame rate.

Therefore, we first determined the maximum connection establishment rate of *Test System 1* shown in Fig. 7.

It is important that the measurement script remotely started and stopped Jool on the DUT before and after each test in order to delete the content of its connection tracking table. For starting Jool, the same commands were used as disclosed in section 6.1. Jool was stopped after each test using the following command:

```
modprobe -r jool
```

As the default timeout of Jool is 5 minutes, we did not need to change it. If one needs to set the timeout, it can be done by the following command:

```
jool global update udp-timeout <value>
```

We limited the possible port number combinations to 4,000,000[7] by using a source port range of [10,000; 49,999] and a destination port range of [80; 179].

We used no background traffic. First, we sent exactly $N = 4,000,000$ number of preliminary frames necessary to fill the state table ($M = 4,000,000$). The global timeout for the preliminary frame sending was $T = 500$ms, and the overall delay before the test phase was calculated as:

$$D = 1000 * \frac{M}{R} + 2 * T \quad (1)$$

We used binary search to determine the *maximum connection establishment rate*, that is, the highest frame rate for the preliminary test, at which all preliminary frames are successfully received by the Responder. The binary search was performed 20 times, and the median, fist percentile, and 99th percentile of the results were determined. In addition to that, we have also determined the dispersion of the results calculated as follows:

$$dispersion = \frac{99th\ perc. - first\ perc.}{median} * 100\% \quad (2)$$

As for frame size to be used, RFC 8219 lists a number of standard frame sizes. We used only the first one of them, 64 bytes for IPv4 and thus 84 bytes for IPv6. Our previous benchmarking experience gained with these test systems shows that the achievable frame rate does not significantly decrease with the frame size, as the bottleneck is the processing power and not the 10Gbps Ethernet [30]. We show an example for testing with a higher frame size in section 6.4.

We have performed the measurements enumerating all possible port number combinations in pseudorandom order. The results are shown in Table 2. Our maximum connection establishment rate results are quite consistent: the first percentile (524,999) and the 99-th percentile (534,814) are quite close to each other.

6.3. Throughput Measurement

Section 5.3 of RFC 8219 requires that all tests be performed with bidirectional traffic. Unidirectional tests are optional, but we performed them, because we were interested, if we could point out any asymmetric behavior of Jool.

[7] Vyacheslav Gapon recommended this number as the size of the connection tracking table for a high-loaded NAT server [29].

Table 2: Maximum connection establishment rate and throughput of the Jool stateful NAT64 implementation ($N = M = 4,000,000$)

Type of measurement	connection est. rate	throughput measurement results bidir.	forward	reverse
Median (fps)	532,368	276,208	523,289	589,493
1st perc. (fps)	524,999	260,470	499,943	561,094
99th p. (fps)	534,814	281,476	544,928	603,132
Dispersion	1.84	7.61	8.60	7.13

As for the parameters, we kept the settings of the connection establishment rate measurement in section 6.2 unless stated otherwise. During the preliminary phase, $R = 500,000$ was used (based on our result in section 6.2).

The results are shown in the last three columns of Table 2. We note that `siitperf` reports the frames/s *per direction* rate, that is, if a bidirectional test is used, then the number of *all* forwarded frames per second is double the reported rate, thus the bidirectional throughput of 276,208fps means a total of 552,416 forwarded frames per second. The 589,493fps unidirectional throughput in the reverse (that is, download) direction is somewhat higher than the 523,289fps in the forward (that is, upload) direction, which seems to be advantageous for the users of a Jool NAT64 gateway. However, the analysis of the results is beyond the scope of our paper. Our measurements aimed to demonstrate the operation of the measurement method.

6.4. Frame Loss Rate Measurement

Frame loss rate measurement is also a part of RFC 8219. It can be performed with the same `siitperf-tp` program using a different shell script, which performs the tests at different frame rates and records the number of successfully received frames.

As an illustration, we have carried out test series using *Test System 1* with the same parameters used for the bidirectional throughput test in section 6.3. Besides using the same 84/64-byte long frames as in all other tests, we have used also 1044/1024-byte long frames. (Another standard frame size, which is significantly higher.) Our results are shown in Fig. 11. The color bars show the median values and the error bars show the first percentile and 99the percentile values. The results are in good agreement with our previous experience [30]: the significantly higher frame size resulted in only a very slightly higher frame loss rate.

6.5. An Initial Performance Estimation of the Stateful Operation of `Siitperf`

We used *Test System 2* for determining the performance limits of `siitperf`. Its topology was very simple as shown in Fig. 8. The two 10GbE interfaces of the Tester were interconnected by a direct cable. Thus, the achievable maximum rates of the looped back Tester were limited by the performance of `siitperf` itself. The hardware and software configuration of p110 was the same as that of p108.

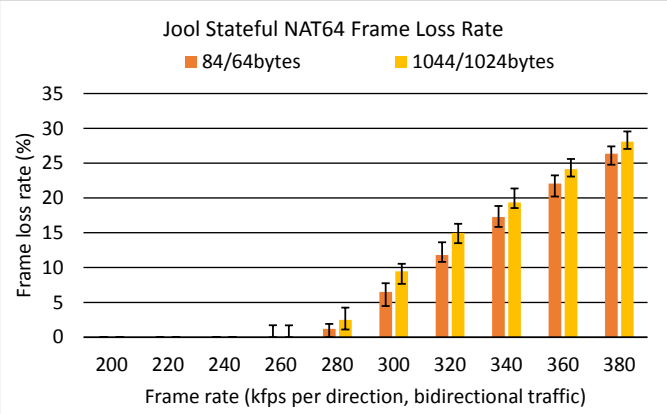

Figure 11: Frame loss rate of the Jool stateful NAT64 implementation as a function of frame rate and frame size using bidirectional traffic

Table 3: Achieved frame rate for maximum connection establishment rate measurements using pre-generated random numbers for port number enumeration

N, M, Port numbers	4,000,000	40,000,000	400,000,000
Median (fps)	7,186,798	7,187,276	7,187,228
1st perc. (fps)	7,183,592	7,187,010	7,187,128
99th perc. (fps)	7,187,256	7,187,501	7,187,400
Dispersion	0.05	0.01	0.00

We note that due to our implementation decision that the Receiver can handle only IPv4 traffic, its "self-test" for performance estimation can only be performed if the Initiator sends IPv4 traffic. However, in the knowledge of the implementation, that is, the `isend()` function first sets pointers to the port numbers depending on the IP version, and then the very same code is used to set the port numbers and to recalculate the UDP checksum, the performance of IPv6 preliminary test frame generation is expected to be very close to that of the IPv4 frames.

We started with the maximum connection establishment rate measurement, using the pseudorandom enumeration of all available port numbers. Unless stated otherwise, the same parameters were used as in section 6.2. The value of N and M, that is, the number of possible port number combinations was increased from 4,000,000 through 40,000,000 to 400,000,000 by using 179, 1079, and 10,079 as the upper limit of the destination port range. The results are shown in Table 3. The results do not decrease with the increase of the number of port number combination at all. It can be easily explained by the fact that the port numbers are pre-generated as described in section 4.4.7, and then the array is read in linear order during the preliminary phase. And the state table of the Responder is written also in linear order. Therefore, their size does not matter: cache prefetching works efficiently.

For the determination of the limits of `siitperf` in throughput testing, we used the same values for port numbers and $R = 7,000,000$. The results are shown in Table 4. This time the values somewhat deteriorate with the increase of the M size of the state table, what can be ex-

Table 4: Achieved frame rate for throughput test (bidirectional traffic) using pseudorandom four tuple selection from the state table of the Tester

N, M, Port numbers	4,000,000	40,000,000	400,000,000
Median (fps)	4,582,440	4,263,139	4,199,651
1st perc. (fps)	4,576,166	4,249,968	4,183,592
99th perc. (fps)	4,583,627	4,264,649	4,201,179
Dispersion	0.16	0.34	0.42

Table 5: Achieved frame rate for throughput test (unidirectional traffic) using pseudorandom four tuple selection from the state table of the Tester ($N = M = 400,000,000$)

Traffic direction	forward	reverse
Median (fps)	6,756,371	4,280,200
1st percentile (fps)	6,756,102	4,265,318
99th percentile (fps)	6,757,568	4,281,251
Dispersion	0.02	0.37

plained by the less and less efficiency of caching due to the pseudorandom 4-tuple selection of the sender of the Responder.

We have also determined the maximum frame rate using unidirectional traffic with $M = 400,000,000$ state table size. The results are shown in Table 5. As expected, the median frame rate in the forward direction (6,756,371fps) is significantly higher than the median frame rate of the bidirectional test (4,199,651fps), because the bottleneck was the reverse direction. The phenomenon that median frame rate in the reverse direction (4,280,200fps) is somewhat higher than the bidirectional one can be explained by the fact that the same NIC is used by the receiver and the sender of the Responder during the bidirectional test. (Theoretically, the reading and writing of the same state table may also have some effect, but we believe that it is not significant due to the very large size of the state table (400,000,000 entries).

We have one further interesting observation: the median frame rate of the throughput in the forward direction (6,756,371fps) is lower than the median of the maximum connection establishment rate (7,187,228fps). That is, `isend()` with pseudorandom enumeration of the port numbers is faster than the stateless `send()` function with RFC 4814 pseudorandom port number generation. The explanation is deliberate: the port numbers for `isend()` are pre-generated, whereas `send()` generates random numbers during the test.

7. Discussion and Future Work

As far as we know, our stateful extension of `siitperf` is the world's first RFC 8219 and RFC 4814 compliant stateful NAT64 / stateful NAT44 tester. Having no sample to follow, we could rely only on our own considerations. Our first test results seem to justify our design concept in various aspects:

1. The usage of the four tuples proved to be a working solution for generating traffic in the direction from the Responder to the Initiator at a sufficiently high frame rate.
2. Separating the preliminary phase and the test phase enabled us to perform a unidirectional test having traffic only from the Responder to the Initiator.
3. Letting the user specify a different frame rate for the preliminary phase than for the test phase enabled us to properly measure both the maximum connection establishment rate and the throughput.
4. Making the extension resilient with several parameters also proved to be useful, e.g. different policies for four tuple selection, resilience regarding the number of preliminary frames, the size of the state table, etc.

Pseudorandom enumeration of all possible port number combinations proved to be a key issue of measuring the maximum connection establishment rate and throughput separately. However, we included linear enumeration of port numbers also in the Internet-Draft [27] as an additional metric. Besides that, linear port number enumeration may also be used for special purposes, like wilfully exhausting the port number range of a stateful NAT64 / NAT44 gateway for simulating a denial of service attack. We plan to use it for testing various NAT64 implementations, how much they are vulnerable to this kind of attack, as we mentioned in [24] and [25].

We are aware that still there are several open questions. For example, in section 6.5, we took the liberty of creating a different number of port number combinations by keeping the source port number range as fixed and increasing the destination port number range tenfold twice. However, we have no idea, how much it is different if we use a source port range of size 10,000 and a destination port range of size 100 versus if we use a source port range of size 40,000 and a destination port range of size 25. The number of possible combinations is 1 million in both cases, but they may result in different performances.

And it was just one example. We expect to gain more experience in stateful testing by carrying out comprehensive benchmarking of various stateful NAT64 implementations like Jool or OpenBSD PF. Our experience may show the need for further developments of `siitperf`.

We believe that having a suitable benchmarking tool is important, but not sufficient. For example, network operator experience regarding the most important parameters of a stateful NAT64 or NAT44 gateway is absolutely necessary for producing usable benchmarking results. Thus, we are looking for partners.

We would be grateful to receive any feedback regarding the theory and practice of stateful testing and also regarding our tool, `siitperf`. Its stateful extension is now available in the "stateful" branch [5], and we plan to merge it into the "master" branch when we consider it to be mature enough.

We are also open to add further functionalities like stateful NAT66 testing if there is user demand for it.

We plan to perform performance optimization when the set of functionalities seems to be stable.

One of the most crucial methodology issues is the problem of using UDP traffic for benchmarking as required by RFC 8219. However, stateful NATxy gateways may handle TCP and UDP "connections" differently. Therefore, it may be necessary to implement testing also with TCP traffic. However, we expect it to be more difficult due to the need for proper handling of TCP connection establishment and termination.

During the first review of this paper, wrote an Internet-Draft [27] about the proposed methodology for stateful NATxy testing and submitted it to the Benchmarking Working Group of IETF. The presentation of its "02" version was received very positively by the session chairs. It is still under development and we hope that one day it may be published as an RFC.

8. Conclusion

We conclude that our efforts were successful in creating the world's first RFC 8219 and RFC 4814 compliant free software stateful NATxy benchmarking tool. Our tests proved that it works correctly and it has high enough performance for benchmarking stateful NAT64 and even stateful NAT44 gateway implementations. We have also advanced the theory of stateful benchmarking by being the first to propose a working solution.

Our future plans include its comprehensive testing, adding further functionalities, and its performance optimization. We also plan to use our new Tester for research in benchmarking methodology issues.

Acknowledgements

The development of `siitperf` and the measurements were carried out remotely using the resources of NICT StarBED, 2-12 Asahidai, Nomi-City, Ishikawa 923-1211, Japan. The author would like to thank Shuuhei Takimoto for the possibility to use StarBED, as well as to Satoru Gonno, Makoto Yoshida, Miku Takuma and Tsukasa Nishita for their help and advice in StarBED usage related issues.

The author thanks Keiichi Shima, Sándor Répás, Ahmed Al-hamadani and Ádám Bazsó for their reading and commenting on the manuscript.

Funding

This work was supported by the Digital Development Center in the national framework GINOP-3.1.1-VEKOP-15-2016-00001—Promotion and support of cooperation between the educational institutions and ICT enterprises.

References

[1] M. Georgescu, L. Pislaru, G. Lencse, Benchmarking methodology for IPv6 transition technologies, IETF RFC 8219 (2017). doi:10.17487/RFC8219.

[2] G. Lencse, Y. Kadobayashi, Comprehensive survey of IPv6 transition technologies: A subjective classification for security analysis, IEICE Transactions on Comminications E102-B (10) (2019) 2021–2035. doi:10.1587/transcom.2018EBR0002.

[3] C. Bao, X. Li, F. Baker, T. Anderson, F. Gont, IP/ICMP translation algorithm, IETF RFC 7915 (2016). doi:10.17487/RFC7915.

[4] M. Bagnulo, P. Matthews, I. Beijnum, Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers, IETF RFC 6146 (2011). doi:10.17487/RFC6146.

[5] G. Lencse, Siitperf: an RFC 8219 compliant SIIT (stateless NAT64) tester written in C++ using DPDK, source code (2019).
URL https://github.com/lencsegabor/siitperf

[6] G. Lencse, Design and implementation of a software tester for benchmarking stateless NAT64 gateways, IEICE Trans. on Commun. E104-B (2) (2021) 128–140. doi:10.1587/transcom.2019EBN0010.

[7] S. Bradner, J. McQuaid, Benchmarking methodology for network interconnect devices, IETF RFC 2544 (1999). doi:10.17487/RFC2544.

[8] D. Newman, T. Player, Hash and stuffing: Overlooked factors in network device benchmarking, IETF RFC 4814 (2008). doi:10.17487/RFC4814.

[9] G. Lencse, Adding RFC 4814 random port feature to siitperf: Design, implementation and performance estimation, Int. J. Advances in Telecomm., Electrotechnics, Signals and Systems 9 (3) (2020) 18–26. doi:DOI:10.11601/ijates.v9i3.291.

[10] NIC Mexico, Jool: SIIT and NAT64, online (2022).
URL http://www.jool.mx/en/

[11] K. J. O. Llanto, W. E. S. Yu, Performance of NAT64 versus NAT44 in the context of IPv6 migration, in: Proc. International Multiconference of Engineers and Computer Scientists 2012 (IMECS 2012), Hong Kong, Hongkong, 2012, pp. 638–645.
URL http://www.iaeng.org/publication/IMECS2012/IMECS2012_pp638-645.pdf

[12] Apache Software Foundation, ab - Apache HTTP server benchmarking tool, online (2022).
URL https://httpd.apache.org/docs/current/programs/ab.html

[13] C. P. Monte, M. I. Robles, G. Mercado, C. Taffernaberry, M. Orbiscay, S. Tobar, R. Moralejo, S. Pérez, Implementation and evaluation of protocols translating methods for IPv4 to IPv6 transition, Journal of Computer Science & Technology 12 (2) (2012) 64–70.
URL http://sedici.unlp.edu.ar/handle/10915/19702

[14] S. Yu, B. E. Carpenter, Measuring IPv4-IPv6 translation techniques, Computer Science Technical Reports (2012-001), Dept. of Computer Science, Univ. of Auckland, Auckland, New Zeeland (2012).
URL http://hdl.handle.net/2292/13586

[15] G. Lencse, S. Répás, Performance analysis and comparison of the TAYGA and of the PF NAT64 implementations, in: Proc. 36th International Conference on Telecommunications and Signal Processing (TSP 2013), Rome, Italy, 2013, pp. 71–76. doi:10.1109/TSP.2013.6613894.

[16] G. Lencse, S. Répás, Performance analysis and comparison of four DNS64 implementations under different free operating systems, Telecommunication Systems 63 (4) (2016) 557–577. doi:10.1007/s11235-016-0142-x.

[17] N. Lutchansky, TAYGA: Simple, no-fuss NAT64 for Linux, online (2011).
URL http://www.litech.org/tayga/

[18] C. Popoviciu, A. Hamza, G. V. de Velde, D. Dugatkin, IPv6 benchmarking methodology for network interconnect devices, IETF RFC 5180 (2008). doi:10.17487/RFC5180.

[19] P. Srisuresh, K. Egevang, Traditional IP network address translator (traditional NAT), IETF RFC 3022 (2001). doi:10.17487/RFC3022.

[20] G. Lencse, Estimation of the port number consumption of web browsing, IEICE Trans. on Commun. E98-B (8) (2015) 15 80–1588. doi:10.1587/transcom.E98.B.1580.

[21] T. Kurahashi, Y. Matsuzaki, T. Sasaki, T. Saito, F. Tsutsuji, Periodic observation report: Internet trends as seen from IIJ infrastructure - 2020 (2021).
URL https://www.iij.ad.jp/en/dev/iir/pdf/iir_vol49_report_EN.pdf

[22] D. Scholz, A look at Intel's dataplane development kit, in: Proc. Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Munich, Germany, 2014, pp. 115–122. doi:10.2313/NET-2014-08-1_15.

[23] D. Liu, H. Deng, NAT46 consideration, expired Internet-Draft (2010).
URL https://tools.ietf.org/html/draft-liu-behave-nat46-02

[24] G. Lencse, Y. Kadobayashi, Methodology for the identification of potential security issues of different IPv6 transition technologies: Threat analysis of DNS64 and stateful NAT64, Computers and Security 77 (1) (2018) 397–411. doi:10.1016/j.cose.2018.04.012.

[25] A. Al-Azzawi, G. Lencse, Identification of the possible security issues of the 464XLAT IPv6 transition technology, Infocommunications Journal 13 (4) (2021) 10–18. doi:10.36244/ICJ.2021.4.2.

[26] R. Durstenfeld, Algorithm 235: Random permutation, Communications of the ACM 7 (7) (1964) 420. doi:10.1145/364520.364540.

[27] G. Lencse, K. Shima, Benchmarking methodology for stateful NATxy gateways using RFC 4814 pseudorandom port numbers, active Internet-Draft (2021).
URL https://datatracker.ietf.org/doc/html/draft-lencse-bmwg-benchmarking-stateful-02

[28] G. Lencse, Scalability of IPv6 transition technologies for IPv4aaS, active Internet-Draft (2022).
URL https://datatracker.ietf.org/doc/html/draft-lencse-v6ops-transition-scalability-01

[29] V. Gapon, Tuning nf_conntrack, personal blog (2019).
URL https://ixnfo.com/en/tuning-nf_conntrack.html

[30] G. Lencse, K. Shima, Performance analysis of SIIT implementations: Testing and improving the methodology, Computer Communications 156 (1) (2020) 54–67. doi:10.1016/j.comcom.2020.03.034.

About author

Gábor Lencse received 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 is also a part time Senior Research Fellow at the Department of Networked Systems and Services, Budapest University of Technology and Economics, Budapest, Hungary since 2005.

His research interests include the performance and security analysis of IPv6 transition technologies. He is a co-author of RFC 8219.