Review version

Design and Implementation of a Software Tester for Benchmarking Stateful NAT64 Gateways: Theory and Practice of Extending Siitperf for Stateful Tests

Gábor LENCSE[†a]

SUMMARY 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 NAT64 gateways. 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 NAT64 and stateful NAT44 (also called NAPT: Network Address and Port Translation) 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 a stateful NAT44 gateway. 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 IPv4 NAPT (Network Address and Port Translation) gateways can be benchmarked using bidirectional traffic with random port numbers.

The remainder of this paper is organized as follows. Section 2 contains a general discussion, how stateful NAT (not necessarily NAT64) 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 presents our functional and performance tests and their results. Section 6 provides a discussion and highlights our plans for further tests, development, performance optimization and research on benchmarking methodology issues. Section 7 gives our conclusions.

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

2.1. 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

Manuscript received April 7, 2021.

†The author is with the Department of Telecommunications, Széchenyi István University, 1 Egyetem tér, H-9026 Győr, Hungary.
a) E-mail: lencse@sze.hu

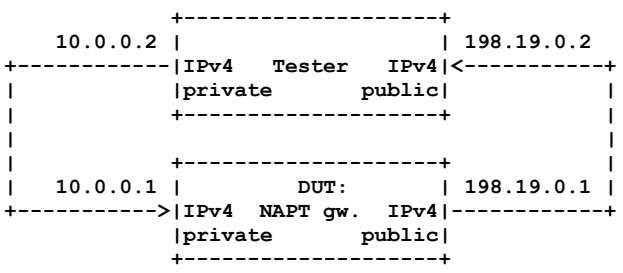

Fig. 1 NAPT gateway test setup (based on RFC 2544).

Address and Port Translation, please refer to Section 2.2 of RFC 3033 [10], 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 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 considered as 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 6.

Let the *connection tracking table* of the NAPT gateway be empty at the beginning of testing, and let the NAPT gateway 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 [11] 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.

2.2 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 the 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 encoded in the statistics. However, several different questions arise, for example:

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 with 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 Siiperf

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) [13] 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 60s 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 parameters that are constant (e.g. IP addresses, MAC addresses, etc.) are supplied in the **`siitperf.conf`** configuration file.

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

[1] Please refer to the report of *Internet Initiative Japan* [12] for a particular observation of the popularity of the different protocols.

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 No.	IP version Left	IP version Right	Type of the DUT	IP addresses used by the Left Sender source	IP addresses used by the Left Sender destination	IP addresses used by the Right Sender source	IP addresses used by the Right Sender 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**

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 follow 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 of test frames on each side.

RFC 8219 also requires that besides the traffic that is translated (we called it as "foreground traffic"), tests should also use non-translated native IPv6 traffic (we called it as "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 # No. of Left side netw.
Num-R-Nets 1 # No. of Right side netw.
```

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 very much 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 not much 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 only 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 pre-generated. 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

As we designed a functional extension of **`siitperf`**, we considered 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 [14] 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*. The Initiator resides on the “private” 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 Responder receives and processes the test frames as needed[2] 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, whereas the Initiator simply acts like[3] the sender and receiver of the original **`siitperf`**.

As the Initiator is completely free to use any source and destination port number combinations during the testing 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.

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 state 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 overwritten 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 NAT 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 5.2, how **`siitperf`** supports the measurement of the maximum connection establishment rate of a stateful device.

4.3.3 Enumeration of Port Numbers

As for the Initiator, its sender could theoretically work the same as the sender of the stateless tester. However, we considered it useful to be able to efficiently exhaust the set of possible port number combinations for two reasons:

- We planned to use this function for surely loading all possibly port number combinations to both the state table of the Responder and to the connection tracking table of the DUT using a minimum *N* number of preliminary test frames. (It was important both from execution time point of view and timeout time point of view.)
- We wanted to create a tool 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 [15] and [16].

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.

---

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

[3] This is a first approximation. Please refer to Subsection 4.3.3 for a refinement.

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 has some consequences for, how the Responder should use certain input parameters regarding foreground traffic[4]:

- It should simply ignore the port number ranges and the number of destination networks specified in the configuration file for the given direction.
- It should reinterpret 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.
In theory, one could say that case 3 is the true spirit of RFC 4814, whereas cases 1 and 2 are computationally less expensive alternatives. However, there is a practical consideration that makes at least one of them a must. We discuss it in Section 4.3.6.
As the above-mentioned configuration file parameters specifying the behavior of the source and destination port numbers may be set differently, and using them would be confusing for the user if **`siitperf`** used one of them and ignored the other one (e.g. source or destination), thus we decided to introduce a new parameter (disclosed in the next subsection).

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
```

Its values have the following meanings:

0 The original operation of **`siitperf`** is kept, the port numbers behave as usual.
1 The port numbers are enumerated in increasing order (source port number is the low order counter and destination port number is high order counter), but the source and destination port numbers are limited to their specified ranges.

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) – 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

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

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. The Responder must surely avoid this situation. Taking the entries of the state table in increasing or decreasing order can be a good guarantee to send only frames belonging to an existing connection, if the frame rate is high enough to scan through the state table within timeout time. (Of course, care must be taken for the timeout also during the preliminary phase, please see our example calculation in Section 5.3.)
If bidirectional traffic is used, then *fast enough* refilling of the state table of the Responder may guarantee that random four tuple selection may find only valid four tuples in it.

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 significant a 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. 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 N is set to a higher value than that, then some of the values will be repeated. Port number enumeration is supported only in the case, when the number of destination networks is set to 1.

The same sender function is called 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 reports 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.
Implementation detail: when the receiver of the Responder is executed, it extracts its input parameters and checks the value of the number of valid entries in the state table. If its value is zero, it indicates that the execution happens in the preliminary phase, and in this case, the receiver allocates NUMA local memory for the state table. It also reports the number of valid entries in the state table at the end of its execution. The non-zero value of the number of valid entries indicates the second execution, when the state table is reused and its entries are overwritten from index 0.
We also 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.
We are aware that we could have implemented an Initiator for the latency tester that supports port number enumeration to provide a completely orthogonal set of functionalities, but we decided not to do that to save time. Thus, currently 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.)
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.

4.4.5 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 currently port number enumeration is not supported in the test phase.

5. 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,
3. to make an initial performance assessment of the stateful operation of **siitperf**.

In our tests, we reused our test systems built in NICT StarBED, Japan for stateless NAT64 measurements [17] and also used for testing the random port feature of **siitperf** [9]. We have also reused the texts of our open access papers [17] and [9] in the description of our test systems (with updates).
We used three Dell PowerEdge C6220 servers with two 2GHz Intel Xeon E5-2650 CPUs having 8 cores each, 128 GB 1333 MHz DDR3 SDRAM and Intel 10G dual-port X520 network adapters. Hyper-threading was

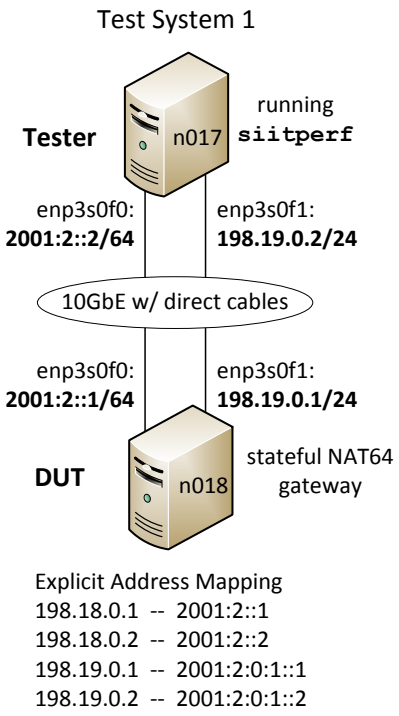

Fig. 2 Test system for the demonstration of the operation of stateful NAT64 tests.

switched off and the clock frequency of all servers was set to 2 GHz (fixed), because we knew from our previous benchmarking experience [18] that they could cause *scattered measurement results*. (We mean under scattered measurement results that the results of the 20 repetitions of the measurements are significantly different.) We aimed to eliminate all circumstances that could cause scattered measurement results.

The Debian Linux operating system was updated to version 9.13 on all three computers. The Linux kernel version was: 4.9.0-4-amd64. The DPDK version was 16.11.11-1+deb9u2.

We used three very similar test setups with somewhat different goals. The aim of *Test System 1* was to demonstrate the operation of a stateful NAT64 measurement. *Test System 2* was used to perform several actual stateful NAT44 measurements. *Test System 3* served an initial performance estimation of **`siitperf`**. The first two of them were identical regarding their hardware; they differed only in the configuration of their software.

Except for the demonstration of the operation of a stateful NAT64 measurement, we used stateful NAT44 for all other tests. We had two reasons for that:

- We wanted to test **`siitperf`** in a situation, where the DUT can achieve high throughput. (We tested the random port feature using IPv4 for the same reason in [9].)
- We plan to perform a comprehensive NAT64 benchmarking and we wanted to avoid possible copyright issues (might arise from publishing the same results in two different papers).

5.1. 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. 2. 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. Stateful NAT64 was implemented in two steps using stateless NAT64 plus stateful NAT44:

1. Stateless NAT64 was implemented by the **`tayga`** stateless NAT64 implementation [19].
2. Stateful NAT44 was implemented by **`iptables`**.

As for **`tayga`**, we have reused our previous Explicit Address Mapping (RFC 7757 [20]) settings [17] as shown at the bottom of Fig. 2. Please refer to the appendix of our open access paper [17] for the detailed settings of **`tayga`**.

As for **`iptables`**, we used the following command:

```
iptables -t nat -A POSTROUTING -o enp3s0f1
-j MASQUERADE
```

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
Responder-ports 3 # 4-tuples random select
```

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*=2000ms. (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: **enp3s0f0** and **enp3s0f1**. They are shown in Fig. 3 and Fig. 4. 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.)

Frames 3-6 are the foreground preliminary frames. In Fig. 3, the 2001:2:0:1::2 IPv6 destination address represents the 198.19.0.2 IPv4 address shown in Fig. 4 as the destination address. The 2001:2::2 source IPv6 address was first mapped to 198.18.0.2 by **tayga**, and then **iptables** replaced it by 198.19.0.1. Port number enumeration can also be observed.

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. Port number enumeration can be observed in the 4 foreground frames from Initiator to the Responder. The port numbers of the 4 foreground frames from the Responder to the Initiator are pseudorandom in the [10000, 10003] range, due to the pseudorandom selection of the four tuples.

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 tuple and uses its elements for traffic generation.

```
 1 0.000000000 fe80::a236:9fff:fe0e:a2c4 → ff02::16 ICMPv6 150 Multicast Listener Report Message v2
 2 0.787987018 fe80::a236:9fff:fe0e:a2c4 → ff02::16 ICMPv6 150 Multicast Listener Report Message v2
 3 2.060717757    2001:2::2 → 2001:2:0:1::2 UDP 80 10000 → 80 Len=18
 4 2.260718236    2001:2::2 → 2001:2:0:1::2 UDP 80 10001 → 80 Len=18
 5 2.460718536    2001:2::2 → 2001:2:0:1::2 UDP 80 10002 → 80 Len=18
 6 2.660724190    2001:2::2 → 2001:2:0:1::2 UDP 80 10003 → 80 Len=18
 7 2.860727275    2001:2::2 → 2001:2:0:8000::2 UDP 80 13787 → 157 Len=18
 8 4.060774275    2001:2::2 → 2001:2:0:1::2 UDP 80 10000 → 80 Len=18
 9 4.060806550 2001:2:0:1::2 → 2001:2::2    UDP 80 80 → 10003 Len=18
10 4.260758051    2001:2::2 → 2001:2:0:1::2 UDP 80 10001 → 80 Len=18
11 4.260817745 2001:2:0:1::2 → 2001:2::2    UDP 80 80 → 10000 Len=18
12 4.460760118    2001:2::2 → 2001:2:0:1::2 UDP 80 10002 → 80 Len=18
13 4.460819319 2001:2:0:1::2 → 2001:2::2    UDP 80 80 → 10002 Len=18
14 4.660760326    2001:2::2 → 2001:2:0:1::2 UDP 80 10003 → 80 Len=18
15 4.660820912 2001:2:0:1::2 → 2001:2::2    UDP 80 80 → 10002 Len=18
16 4.860769785    2001:2::2 → 2001:2:0:8000::2 UDP 80 21136 → 86 Len=18
17 4.860778173 2001:2:0:8000::2 → 2001:2::2    UDP 80 28744 → 41552 Len=18
```

Fig. 3 The **tshark** capture of a stateful NAT64 test on the **enp3s0f0** interface of the DUT.

```
 1 0.000000000 fe80::a236:9fff:fe0e:a2c6 → ff02::16 ICMPv6 150 Multicast Listener Report Message v2
 2 0.731995970 fe80::a236:9fff:fe0e:a2c6 → ff02::16 ICMPv6 150 Multicast Listener Report Message v2
 3 2.036790798   198.19.0.1 → 198.19.0.2   UDP 60 10000 → 80 Len=18
 4 2.236775622   198.19.0.1 → 198.19.0.2   UDP 60 10001 → 80 Len=18
 5 2.436775998   198.19.0.1 → 198.19.0.2   UDP 60 10002 → 80 Len=18
 6 2.636781951   198.19.0.1 → 198.19.0.2   UDP 60 10003 → 80 Len=18
 7 2.836746677    2001:2::2 → 2001:2:0:8000::2 UDP 80 13787 → 157 Len=18
 8 4.036763279   198.19.0.2 → 198.19.0.1   UDP 60 80 → 10003 Len=18
 9 4.036833787   198.19.0.1 → 198.19.0.2   UDP 60 10000 → 80 Len=18
10 4.236767412   198.19.0.2 → 198.19.0.1   UDP 60 80 → 10000 Len=18
11 4.236812336   198.19.0.1 → 198.19.0.2   UDP 60 10001 → 80 Len=18
12 4.436771779   198.19.0.2 → 198.19.0.1   UDP 60 80 → 10002 Len=18
13 4.436814380   198.19.0.1 → 198.19.0.2   UDP 60 10002 → 80 Len=18
14 4.636770939   198.19.0.2 → 198.19.0.1   UDP 60 80 → 10002 Len=18
15 4.636815797   198.19.0.1 → 198.19.0.2   UDP 60 10003 → 80 Len=18
16 4.836778418 2001:2:0:8000::2 → 2001:2::2    UDP 80 28744 → 41552 Len=18
17 4.836789464    2001:2::2 → 2001:2:0:8000::2 UDP 80 21136 → 86 Len=18
```

Fig. 4 The **tshark** capture of a stateful NAT64 test on the **enp3s0f1** interface of the DUT.

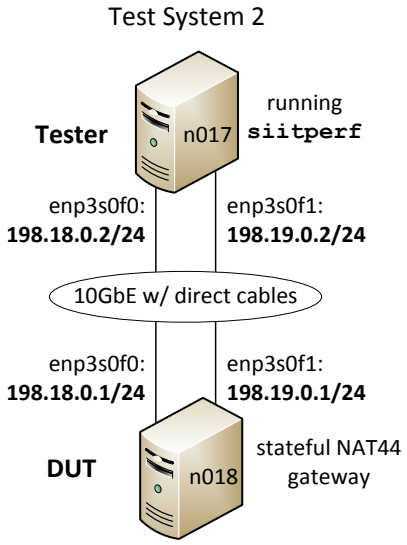

Fig. 5 Test system performing stateful NAT44 benchmarking measurements.

5.2. Measurement of the Maximum Connection Establishment Rate of a Stateful NAT44 Gateway.

Before an actual stateful NAT44 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 2 shown in Fig. 5.

Regarding Test System 2, it is an important condition that all cores of the second CPU of the DUT were switched off using the **`maxcpus=8`** kernel parameter to avoid NUMA issues. (In these servers, cores 0-7 belong to NUMA node 0 and cores 8-15 belong to NUMA node 1.)

The various settings of the NAT44 gateway may drastically influence its throughput. We wanted to imitate the conditions of an ISP, therefore, we set the parameters of the connection tracking table following the recommendations of Vyacheslav Gapon for a high-loaded NAT server [21]. Namely, the **`nf_conntrack_max`** and **`hashsize`** parameters were set to 4,194,304 and 524,288, respectively.

It is important that the measurement script remotely started and stopped **`iptables`** on the DUT before and after each test in order to delete the content of its connection tracking table.

To avoid the exhaustion of the connection tracking table during the tests, we limited the possible port number combinations to 4,000,000 by using source port range of [10,000; 49,999] and 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=500ms, and the delay of the preliminary phase was calculated as:

$$D=1000*M/N+2*T=2000 \quad (1)$$

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 and processed by the Responder. The binary search was performed 20 times, and the median, minimum and maximum were determined. In addition to that, we have also determined the dispersion of the results calculated as follows:

$$dispersion = \frac{max - min}{median} \cdot 100\% \quad (2)$$

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. 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 [17]. We show an example for testing with higher a frame size in Section 5.4.

We have performed the measurements both using pseudorandom port numbers and using port number enumeration. The results are shown in Table 2. Whereas the median of the maximum connection establishment rate using pseudorandom port numbers is 1,406,230fps, the median of the maximum connection establishment rate using port number enumeration is only 669,587fps. The second one is less than half of the previous one, which needs an explanation. We were aware that when we used random port numbers, then some of the combinations were repeated and even though the state table of the Tester was filled, the connection tracking table of the DUT contained fewer elements: its number was around 2.5 million, instead of 4 million. We have repeated the test with N=M=40,000,000 so that the state table of the DUT be filled up, too. Its result is shown in the last column of Table 2. Compared to the first case, the median value decreased by less than 10% from 1,406,230fps to 1,271,023fps. It means that the root

Table 2 Maximum connection establishment rate of **`iptables`** stateful NAT44

Port numbers are	random	enumerated	random, M: 10x
Median (fps)	1,406,320	669,587	1,271,023
1st perc. (fps)	1,400,377	664,030	1,245,115
99th perc. (fps)	1,410,172	675,941	1,291,504
Dispersion	0.70	1.78	3.65

Table 3 The throughput of **iptables** stateful NAT44

Traffic	bidir. rp: 3	bidir. rp: 1	forward	reverse
Median (fps)	932,919	875,861	1,891,016	1,498,951
min (fps)	926,945	872,046	1,749,999	1,453,124
max (fps)	936,891	880,500	1,906,251	1,531,495
Dispersion	1.07	0.97	8.26	5.23

cause of the much lower rate was not the higher number of elements in the connection tracking table of the DUT, but the port number enumeration. It is beyond the scope of our paper to investigate why the usage of increasing port numbers deteriorate the performance of **iptables** to such an extent, but it is an important lesson for us that the random or ordered nature of the port numbers of the consecutive packets may have a significant effect on the maximum connection establishment rate. Thus this observation limits the applicability of such tests.

5.3. Throughput Measurement of a Stateful NAT44 Gateway.

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 behaviour of **iptables**.

As for the parameters, we kept the settings of the connection establishment rate measurements in Section 5.2 unless stated otherwise. The *N* number of preliminary frames and the *M* size of the state table requires some discussion. If the connection tracking table of the DUT is not filled during the preliminary phase, its filling is completed during the 60s long throughput test, if there is traffic in the forward direction. It will not happen if a unidirectional test is performed in the reverse direction. Therefore, it is desirable to fill the connection tracking table of the DUT in the preliminary phase as much as possible. However, we need to consider the timeout limitations, too. The timeout of the UDP connections is 30 seconds.

First, we have performed some preliminary tests allowing that the connection tracking table was filled only partially (about 2.5 million entries) to gain some insight regarding the order of magnitude of the throughput of the DUT. We have found that the bidirectional throughput value fell into the [800,000fps, 1,000,000fps] interval, and the unidirectional throughput values fell into the [1,000,000fps, 2,000,000fps] interval.

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.

As for the bidirectional test, we have chosen 1,400,000fps as the upper bound of the binary search,

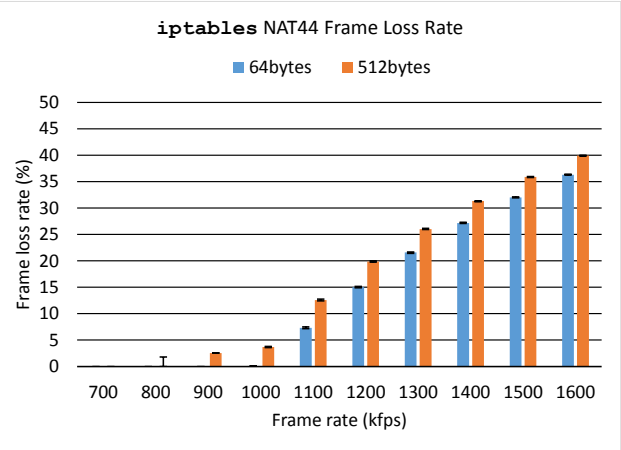

Fig. 6 Frame loss rate of **siitperf** NAT44 as a function of frame rate and frame size using bidirectional traffic

and we have considered 700,000fps as the lowest frame rate to be used. We have chosen N=M=20,000,000. Even when sending at only 700,000fps rate, the Responder can go through the state table in 28.6s. Based on our results for the maximum connection establishment rate, we have chosen R=1,000,000 to leave some performance reserve. In our case, the delay caused by the preliminary phase is *D*=21s, which is deliberately not the bottleneck, but in other cases one should also consider it, when calculating the limits of the timeout. As for the unidirectional tests, we used 2,000,000fps as the upper bound of the binary search.

We have performed the bidirectional test in two ways to check, if it makes a difference. First, we used pseudorandom four tuple selection at the Responder (by setting **Responder-ports** to 3), and then we used linear scanning of the state table in increasing order (**Responder-ports**: 1). The results are shown in Table 3. The difference is quite visible: the second setting reduces the median throughput from 932,919fps to 875,861 by 6.1%. As for the unidirectional tests, this setting was redundant during the test in the forward direction (as there was no traffic from the responder to the initiator), and linear scanning was the only workable solution during the test in the reverse direction. We have found a slight asymmetry: whereas the median throughput was 1,891,016fps in the forward direction, it was only 1,498,951fps in the reverse direction (about 20.7% less).

5.4. 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

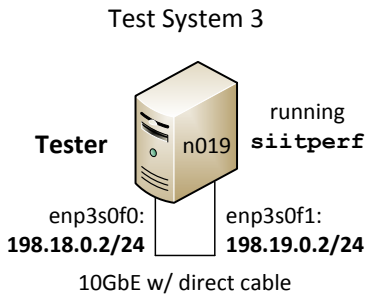

Fig. 7 Test system for determining the performance limits of **siitperf**.

Table 4 Achieved frame rate for maximum connection establishment rate measurement

Port numbers	random	enumerated
Median (fps)	6,251,382	6,701,677
1st perc. (fps)	6,249,999	6,687,499
99th perc. (fps)	6,251,709	6,702,881
Dispersion	0.03	0.23

Test System 2 with the same parameters used for the bidirectional throughput test using random four tuple selection in Section 5.3. Besides using the same 64-byte long frames as in all other tests, we have used also 512-byte long frames. (This standard frame size was selected to be significantly larger, but still small enough to prevent the 10Gbps Ethernet from being a bottleneck.) Our results are shown in Fig. 6. The colour bars show the median values and the (usually invisible) error bars show the minimum and maximum values. The results for 64-byte frames are in a good agreement with our throughput measurement results: there is no frame loss up to 900kfps rate, and though the frame loss rate is invisible in the figure, the median of the frame loss is 0.08% at 1000kfps frame rate, and it visibly grows from 1100kfps frame rate. As for the 512-byte frames, an error bar appears at 800kfps, showing that the maximum of the frame loss rate was 1.8% and the frame loss rate visibly grows from 900kfps frame rate. Except for the case of 512-byte frames at 800kfps frame rate, the results are very stable: the error bars are practically invisible in all other cases.

5.5. An Initial Performance Estimation of the Stateful Operation of **Siitperf**.

We used Test System 3 for determining the performance limits of **siitperf**. Its topology was very simple as shown in Fig. 7. The two 10GbE interfaces of the Tester were interconnected by a direct cable. Thus, the performance of the looped back Tester was limited by the performance of **siitperf** itself.

Table 5 Achieved frame rate for throughput test with 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)	4,045,161	3,801,450	3,733,430
Minimum (fps)	4,039,061	3,781,249	3,703,001
Maximum (fps)	4,048,891	3,802,918	3,742,249
Dispersion	0.24	0.57	1.05

Table 6 Achieved frame rate for throughput test with linear scanning of the state table of the Tester

N, M, Port numbers	4,000,000	40,000,000	400,000,000
Median (fps)	5,035,116	5,034,443	5,034,109
Minimum (fps)	5,033,136	5,032,957	5,033,202
Maximum (fps)	5,039,063	5,035,652	5,035,284
Dispersion	0.12	0.05	0.04

Table 7 Achieved frame rate of **siitperf-pdv** with state table size M=4,000,000

four tuple selection	random	linear scan
Median (fps)	3,643,421	4,620,949
Minimum (fps)	3,642,328	4,619,952
Maximum (fps)	3,644,532	4,623,047
Dispersion	0.06	0.07

Unless stated otherwise, we used the settings of the connection establishment rate measurements in Section 5.2.

First, we tested the preliminary phase performance both using random ports and port number enumeration. The results are shown in Table 4. As we expected, **siitperf** kept its high performance.

Based on the results, we used R=6,000,000 for the throughput tests.

We performed two sets of throughput tests. In both series, pseudorandom port numbers were used by the sender of the Initiator. The value of N and M, as well as the size of the port range 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. In the first series, the sender of the Responder used pseudorandom four tuple selection, whereas in the second series, the Responder used the four tuples from its state table in increasing order.

The results of the first measurement series are shown in Table 5. Throughput as a function of the M size of the state table shows a slightly decreasing tendency. We attribute it to the fact that caching becomes less and less effective as the size of the state table grows. We can support it with the results of the second series in Table 6. On the one hand, the results are higher than the results in Table 5, also because in the second series there was no need to generate a random number. However, on the other hand, the results are not degrading here, because

the access of the elements of the state table happened in an increasing order, which made caching and likely also cache pre-fetching effective.
As discussed in [9], the low number of latency frames do not influence the performance of **siitperf**, therefore, there was no need to measure the performance of **siitperf-lat**.
As for the performance of **siitperf-pdv**, we have checked it at state table size M=4,000,000 using both random and linear four tuple selection. The results are shown in Table 7. As we expected, maintaining a 64-bit counter in each frame has its costs, and linear four tuple selection gives higher performance than random.

6. 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 preliminary phase and 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 at the preliminary phase enabled us to properly measure the throughput in the case if it is higher than the maximum connection establishment rate (as seen in Section 5.3).
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.

Port number enumeration was another concept, which we expected to be of practical use in benchmarking. We expected it to enable us efficiently filling up the connection tracking table of the DUT and the state table of the Responder. On the one hand, our results in Section 5.3 justified our intention to save timeout time. However, on the other hand, our results regarding the maximum connection establishment rate in Section 5.2 have shown that the random or ordered nature of the port numbers of the consecutive packets may have a significant effect on the maximum connection establishment rate, hence, the performance of a stateful gateway. Therefore, we had to use a more complicated method in Section 5.3 to fill up the state table of the DUT. However, we still consider it as a useful feature of **siitperf**, which may be applied 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 [15] and [16].
We are aware that still there are several open questions. For example, in Section 5.4, we took the liberty of creating 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 performance.
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 matured 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 seem to be stable.

6. Conclusions

We conclude that our efforts were successful in creating the world's first RFC 8219 and RFC 4814 compliant free software stateful NAT64/NAT44 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.
One of the most crucial methodology issues is the problem of using UDP traffic for benchmarking as

required by RFC 8219. This can be a serious problem for two reasons:

- The default timeout values of **`iptables`** are different for TCP and UDP "connections".
- The handling of TCP and UDP "connections" is very likely also different.

Therefore, we believe that it is 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.

We also plan to write an Internet Draft about the proposed methodology for stateful testing and submit it to the Benchmarking Working Group of IETF.

Acknowledgments

The development of **`siitperf`** and the measurements were carried out by 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 and Miku Takuma for their help and advice in StarBED usage related issues.

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

References

[1] M. Georgescu, L. Pislaru and G. Lencse, "Benchmarking Methodology for IPv6 Transition Technologies", *IETF RFC 8219*, Aug. 2017, DOI: 10.17487/RFC8219

[2] G. Lencse and Y. Kadobayashi, "Comprehensive survey of IPv6 transition technologies: A subjective classification for security analysis", *IEICE Trans. Commun.*, vol. E102-B, no. 10, pp. 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*, June 2016, DOI: 10.17487/RFC7915

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

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

[6] G. Lencse, "Design and Implementation of a Software Tester for Benchmarking Stateless NAT64 Gateways", IEICE Trans. Commun., vol. E104-B, no.2, pp. 128–140. Feb. 2021. DOI: 10.1587/transcom.2019EBN0010

[7] S. Bradner and J. McQuaid, "Benchmarking methodology for network interconnect devices", *IETF RFC 2544*, March 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*, vol. 9, no. 3, pp. 18–26, 2020, DOI: 10.11601/ijates.v9i3.291

[10] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)", *IETF RFC 3022*, January 2001. DOI: 10.17487/RFC3022

[11] G. Lencse, "Estimation of the Port Number Consumption of Web Browsing", *IEICE Trans. Commun.*, vol. E98-B, no. 8. pp. 1580–1588. Aug. 2015. DOI: 10.1587/transcom.E98.B.1580

[12] T. Kurahashi, Y. Matsuzaki, T. Sasaki, T. Saito, F. Tsutsuji, "Periodic observation report: Internet trends as seen from IIJ infrastructure – 2020", *Internet Infrastructure Review*, vol. 49, December 25, 2020. https://www.iij.ad.jp/en/dev/iir/pdf/iir_vol49_report_EN.pdf

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

[14] D. Liu and H. Deng, "NAT46 consideration," expired Internet Draft, https://tools.ietf.org/html/draft-liu-behave-nat46-02

[15] G. Lencse and Y. Kadobayashi, "Methodology for the identification of potential security issues of different IPv6 transition technologies: Threat analysis of DNS64 and stateful NAT64", *Computers & Security* (Elsevier), vol. 77, no. 1, pp. 397-411, August 1, 2018, DOI: 10.1016/j.cose.2018.04.012

[16] A. Al-Azzawi and G. Lencse, "Towards the Identification of the Possible Security Issues of the 464XLAT IPv6 Transition Technology", *43rd International Conference on Telecommunications and Signal Processing (TSP 2020)*, Milan, Italy, July 7-9, 2020, pp. 439-444.

[17] G. Lencse, K. Shima, "Performance Analysis of SIIT Implementations: Testing and Improving the Methodology", *Comput. Commun.*, vol. 156, pp. 54–67, April 15, 2020, DOI: 10.1016/j.comcom.2020.03.034

[18] Lencse G., Kadobayashi Y., "Benchmarking DNS64 implementations: Theory and practice", *Comput. Commun.*, vol. 127, 2018, pp. 61–74, DOI: 10.1016/j.comcom.2018.05.005

[19] Lutchansky, N. (2011). TAYGA: Simple, no-fuss NAT64 for Linux, http://www.litech.org/tayga/ Accessed March 1, 2021.

[20] Anderson, T., Potter, A. L. (2016). Explicit address mappings for stateless IP/ICMP Translation, IETF RFC 7757, DOI: 10.17487/RFC7757

[21] V. Gapon, "Tuning nf_conntrack", personal blog, https://ixnfo.com/en/tuning-nf_conntrack.html Accessed March 1, 2021.

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