Adding RFC 4814 Random Port Feature to Siitperf: Design, Implementation and Performance Estimation

G. Lencse

***Abstract*—Siitperf is the World's first free software RFC 8219 compliant SIIT (also called stateless NAT64) tester written in C++ using DPDK, which is also suitable for benchmarking IPv4 / IPv6 network interconnect devices in RFC 2544 / RFC 5180 compliant ways. Originally, siitperf followed RFC 2544 Appendix C.2.6.4 test frame format resulting in "hard coded" source and destination UDP port numbers. RFC 4814 Section 4.5 recommended random, uniformly distributed source and destination port numbers, which can make a very significant difference, when the DUT (Device Under Test) has multiple CPU cores, what is very common today. Therefore, adding this feature to siitperf is essential to be able to produce meaningful benchmarking results. In this paper, we disclose the design, implementation and performance estimation of this extension of siitperf.**

***Keywords*—benchmarking, frame loss rate, latency, packet delay variation, port number, SIIT, throughput.**

I. Introduction

RFC 8219 [1] has defined a comprehensive benchmarking methodology for *IPv6 transition technologies* [2] by classifying the high number of IPv6 transition technologies into a small number of categories and defining measurement procedures for each category. For example, SIIT [3] (also called stateless NAT64) belongs to the category of *single translation technologies*. We have shown that legacy RFC 2455 [4] / RFC 5180 [5] compliant network performance testers can be used to perform RFC 8219 compliant *throughput* and *frame loss rate* tests of SIIT gateways with some tricks [6]. However, the applicability of legacy testers is limited, because the *latency* measurement procedure has been redefined in RFC 8219 (to use at least 500 timestamps instead of a single one) and RFC 8219 introduced *PDV* (Packet Delay Variation) tests. Therefore, new, RFC 8219 compliant testers are needed. As far as we know, our **`siitperf`** [7] is the World's first free software RFC 8219 compliant SIIT tester. It is available under GPLv3 license from GitHub [8]. We have implemented it in C++ using Intel's DPDK (Data Plane Development Kit [9]) to achieve high enough performance. During its design, we have made several generalizations to make our tester flexible, for example, the IP versions of the two sides may be set independently from each other, thus **`siitperf`** can also be used for benchmarking IPv4 / IPv6 network interconnect devices (e.g. routers) in RFC 2544 / RFC 5180 compliant ways. However, being not aware of RFC 4814 [10], we have closely followed the *test frame format* originally defined in Appendix C.2.6.4 of RFC 2544 and implicitly reused in RFC 5180 and RFC 8219, which has defined "hard coded" source and destination UDP port numbers.

On the one hand, the usage of fixed port numbers (together with fixed IP addresses) allows the reuse of the very same test frames, which can be a performance advantage for software testers. However, on the other hand, our SIIT benchmarking experience showed that the usage of fixed test frames resulted in a situation, were only two CPU cores were used[1] (one core for each direction) from the several cores of a the two CPUs of the computer used as the DUT (currently: SIIT gateway) [11]. We believe that the results of such measurements do not reflect the real life performance of a multi-core DUT well enough, because a high number of different IP addresses and different port numbers occur in a real life traffic, thus the interrupts are hashed more or less equally to all CPU cores. Therefore, we were planning to use a non-standard solution of increasing the source port numbers one by one, which we have successfully used with the new version of **`dns64perf++`** [12], when it was necessary for benchmarking high performance authoritative DNS servers [13].

Alfred C. Morton, co-chair of the IETF Benchmarking Working Group (BMWG), has advised us about RFC 4814 in his reply to the BMWG mailing list [14]. Section 4.5 of RFC 4814 recommends pseudorandom and uniformly distributed values for both source and destination port numbers. Our current effort aims to extend **`siitperf`** with an RFC 4814 compliant random port feature.

The remainder of this paper is organized as follows. In Section II, we give a very brief overview of **`siitperf`**. In Section III, we disclose our design considerations and most important implementation decisions. In Section IV, we present various performance tests and their results. In Section V, we provide a short discussion of what our results really mean concerning the method, how to use varying port numbers and we also give some directions of future research. Section VI concludes our paper.

Manuscript received September 12, 2020.

G. Lencse is with the Department of Networked System and Services, Budapest University of Technology and Economics, Magyar tudósok körútja 2, H-1117 Budapest, Hungary. (e-mail: lencse@hit.bme.hu).

[1] In fact, we could observe only the interrupts. When the interrupts fully utilized the capacity of the given CPU core, then the CPU core became a bottleneck.

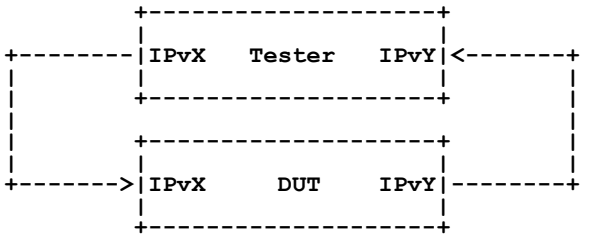

Fig. 1 Single DUT test setup [1].

II. Overview of Siitperf in a Nutshell

We give only a very brief introduction to **siitperf** focusing on the information that is necessary to understand the rest of this paper, because all the details are available in our open access paper [7], the text of which we reused in this section. For even more details, the commented source code of **siitperf** is also available from GitHub [8].

The test and traffic setup of **siitperf** follows the single DUT setup of RFC 8219 shown in Fig. 1. The IP versions of the *left side* and *right side* interfaces of the Tester and of the DUT are *IPvX* and *IPvY*, respectively, where $X,Y \in \{4,6\}$. According to RFC 8219, $X \neq Y$, however **siitperf** allows $X=Y$, too. Thus, besides SIIT gateways, **siitperf** can also be used for benchmarking IPv4 or IPv6 network interconnect devices (e.g. routers). Although the arrows of Fig. 1 would imply unidirectional traffic, testing with bidirectional traffic is required by RFC 8219 and testing with unidirectional traffic is optional. As for naming the directions, we called the direction following the arrows as *forward* direction and the opposite one as *reverse* direction. RFC 8219 requires to use the mixture of translated traffic plus non-translated, native IPv6 traffic (a few different proportions are required). We called the translated traffic as *foreground traffic* and we named the non-translated IPv6 traffic as *background traffic*.

As for the scope of measurements, **siitperf** supports the *throughput*, *frame loss rate*, *latency* and *PDV* (Packet Delay Variation) tests.

Following the requirement inherited from RFC 2544, **siitperf** supports testing with a single source and destination address pair as well as the case, when the destination addresses are random and uniformly distributed over a range of 256 networks. However, the UDP source and destination port numbers are always fixed values following the *test frame format* defined in Appendix C.2.6.4 of RFC 2544.

As for the design and implementation of **siitperf**, we have implemented the core of the measurements in C++ using Intel's DPDK (Data Plane Development Kit [9]). A single execution performs one elementary measurement with some well-defined parameters, and bash shell scripts are used to perform the tests with different parameters. The parameters were divided into two groups: those that do not change during the consecutive executions of **siitperf**, are put into the **siitperf.conf** configuration file, and those that may be changed, are supplied by the bash shell scripts as command line parameters.

As for implementation, **siitperf** is manifested as three similar, but slightly different programs:

- **siitperf-tp** can be used for throughput and frame loss rate measurements (with two different bash shell scripts),
- **siitperf-lat** is for latency measurements,
- **siitperf-pdv** can be used for PDV measurements and also for a special kind of throughput measurements using individual frame timeout, the rationale of which we have shown in [15].

The three programs share the same code base and their operation is also very similar. Our original object oriented design concept is very simple: the **Througput** class is responsible for the majority of the tasks (reading and storing the parameters, controlling and executing the measurement, as well as evaluating its results), and the **Latency** and **Pdv** classes extend it with some special functions.

The control structure of the programs is also very simple: first, the parameters are read from the configuration file and from the command line, then the hardware of the Tester is initialized, and finally, the **measure()** member function of the proper class is called. In the general case, **measure()** starts four threads: one sender and one receiver for each direction. (Unidirectional tests require only a single sender and receiver pair.) The sender and receiver threads are executed by their own CPU cores, which are excluded from the scheduler of the Linux kernel using the **isolcpus** kernel parameter.

During the implementation of **siitperf**, we have encountered the following inconvenient feature of DPDK: the **rte_eal_remote_launch()** function, which we used to start the sender and receiver functions on the appropriate cores, does not allow the execution of non-static member functions. Therefore, the sender and receiver functions are not member functions of the above mentioned three classes but they are standalone functions, and their input parameters are packed into proper data structures.

To achieve as high performance as possible, we used several optimizations:

- Throughput tests send the same pre-generated foreground or background frames. (If multiple destination networks are used, then the frames are pre-generated for all possible destinations.)
- All the special (tagged and numbered) frames for latency measurements are also pre-generated.
- As all the test frames for PDV measurements have unique 64-bit IDs, the pre-generated test frames are modified and their pre-computed UDP checksums are adjusted.

Regarding the modification of the test frames, we have faced with a very strange phenomenon. The official description of **rte_eth_tx_burst()** function says that: "The **rte_eth_tx_burst()** function returns the number of packets it actually sent." However, its detailed description says that:

"For each packet to send, the **rte_eth_tx_burst()** function performs the following operations:

- Pick up the next available descriptor in the transmit ring.

- Free the network buffer previously sent with that descriptor, if any.
- Initialize the transmit descriptor with the information provided in the **`*rte_mbuf`** data structure." [16]

We have found that in fact the **`rte_eth_tx_burst()`** function does not wait until the frames are sent, but it reports the frames as sent, when they are still in the transmit buffer. It also means that if we rewrite a frame right after its sending, we may overwrite it before its actual transmission occurs. Therefore, we used *N* number of copies of each pre-generated frame that we had to modify, and we always used the next copy in a round robin manner.

III. Design and Implementation

A. Requirements of RFC 4814 and our Design Decisions

Section 4.5 of RFC 4814 says: "unless known port numbers are specifically required for a test, it is recommended to use pseudorandom and uniformly distributed values for both source and destination port numbers".

To make **`siitperf`** flexible, we decided to enable the user to make a decision about fixed or varying nature of the source and destination port numbers independently, and to do so for the forward and reverse directions independently, too.

Besides the recommended pseudorandom port numbers, we also opened up the possibility of one by one increasing and decreasing port numbers. Increasing port numbers were chosen as a computationally cheaper alternative to the generation of pseudorandom port numbers and it may also be interesting to scan a given port range. (Decreasing ones were added as an also easy to implement alternative of the increasing ones. For example, the commercial Spirent SPT-N4U Tester also implements them.)

We have chosen a simple encoding of the four possible choices about the port numbers: 0: fixed, 1: increasing, 2: decreasing, 3: pseudorandom. Thus, we added the following new options to the **`siitperf.conf`** configuration file:

```
Fwd-var-sport 0 # forward source ports: fixed
Fwd-var-dport 1 # fwd. dest. ports: incr.
Rev-var-sport 2 # reverse sports: decreasing
Rev-var-dport 3 # rev. dest. ports: random
```

To keep the default behavior of **`siitperf`** compatible with the original one, their default values are 0.

As for the ranges of source port numbers and destination port numbers, section 4.5 of RFC 4814 recommends the ranges of [1024, 65535] and [1, 49151], respectively. Whereas these ranges seem to be logical for the *first UDP datagram* (or the SYN segment of TCP), source and destination ports change their roles in the reply, therefore, in our opinion, forwarding devices (e.g. routers, SIIT gateways, etc.) should be able to handle source and destination ports in the full [1, 65535] (or even [0, 65535]) range. Therefore, we decided to let the user set any values in the range of [0, 65535].

In our sample configuration file, we set the values recommended by RFC 4814 as follows:

```
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
```

As for performance requirements, it was crucial to keep the high performance of **`siitperf`**, because the usage of varying port numbers results in significant increase of the performance of multi core DUTs.

B. Implementation Details

We tried to keep as much as possible from our original performance optimized code. Therefore, we used the same trick as originally with PDV: we pre-generate *N* copies of test frames (to mitigate the rewrite after send problem), compute their (un-complemented) UDP checksums, and modify the pre-generated test frames and their checksums as necessary due to the changing port numbers. We note that the special (tagged and numbered) frames for latency measurements are exceptions: as they are not reused, they exist only in a single instance (and not in *N* copies). We also kept the original code for the case, when the user requires fixed port numbers. It gives us a good basis for comparison of the performance of our new code.

For pseudorandom port number generation, we have chosen the same 64-bit *Mersenne Twister* pseudorandom number generator (**`std::mt19937_64`**), which we already used before for generating random destination networks. Of course, we used a separate instance for every single purpose.

We consider it important, how the modification of the pre-generated frames happens. The type of the frame to be modified can be: an IPv4 or IPv6 foreground frame, a background frame (always IPv6), or an IPv4 or IPv6 latency frame (in the case of latency measurement). The program sets working pointers to the fields to be modified, and the modification is done at a single point independently from the type of frame, which is an important advantage from the viewpoint of testing and maintenance of the source code.

Currently, our new source code is available as the "varport" branch of **`siitperf`** on GitHub [8]. On the long run, we plan to merge it into the master branch.

IV. Various Tests and their Results

A. Measurement Environment

The aims of our measurements were the following ones:

1. To perform the most important benchmarking tests and to examine the effect of the random ports on the benchmarking measurements.
2. To measure the "performance cost" of the varying port numbers, that is, how the maximum achievable frame rate of **`siitperf`** decreases, when random or increasing port numbers are used.
3. To test if there is a benefit in extending the source and destination port number rages to [0, 65535].

To achieve these goals, we needed a test system, where there are no "disturbing factors" like scattered results due to hyper-threading or CPU frequency scaling, performance deviations due to CPU power budget limitations or NUMA (Non-Uniform Memory Access) issues, etc.

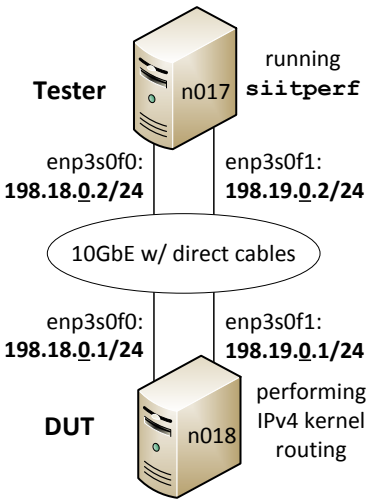

Fig. 2 Test system for benchmarking IPv4 kernel routing.

Based on our previous benchmarking experience, we selected three identical Dell PowerEdge C6220 servers in the NICT StarBED, Japan. They were equipped with two 2GHz Intel Xeon E5-2650 CPUs having 8 cores each, 128GB 1333MHz DDR3 RAM and Intel 10G dual port X520 Ethernet network adapters.

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

To be able to achieve higher frame rates, we benchmarked IPv4 kernel routing (and not SIIT), because our aim was to check and demonstrate the behavior of **`siitperf`** at demanding frame rates. The topology of the test system is shown in Fig. 2. The Tester and the DUT were interconnected by two 10GbE direct cable links. "Turbo Mode" was enabled on the Tester (n017) to have some performance reserve (later it proved to be unnecessary). Turbo Mode was disabled on the DUT (n018), the clock frequency of which was set to fixed 2GHz. All cores of the second CPU of the DUT were switched off using the **`maxcpus=8`** kernel parameter to avoid NUMA issues. (In these computers, cores 0-7 belong to NUMA node 0 and cores 8-15 belong to NUMA node 1.)

We have built another test system for determining the performance limits of **`siitperf`**. Its topology was very simple as shown in Fig. 3. 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. The clock frequency of n019 was set to fixed 2GHz.

Naturally, hyper-threading has been disabled on all three computers.

Both Testers used the **`isolcpus=4,5,6,7`** kernel parameter to reserve the appropriate cores for the four working threads of **`siitperf`**.

The IP addresses of the interfaces of n017 and n018 were set according to Fig. 2 for the tests, when only a single destination network (per direction) was used. When 256 destination networks (per direction) were used, the 256 destination networks were created that the underlined zeros in the IP addresses in Fig. 2, were randomly replaced by one of the numbers from 0 to 255. For these tests, the appropriate IP addresses from each network were assigned to the DUT by a script. As **`siitperf`** currently does not support ARP, the appropriate static ARP entries were set manually in the DUT (using a script).

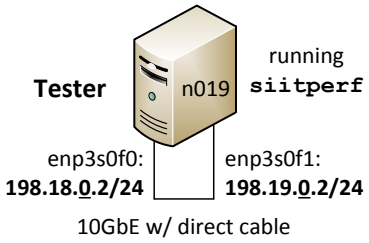

Fig. 3 Test system for determining the performance limits of **`siitperf`**.

Multi-queue receiving (also called Receive-Side Scaling [17]) considers only the source and destination IP addresses in the hash function to distribute the incoming packets into the queues by default, that is, to assign them (including the processing of the interrupts) to the CPU cores. Therefore, we used the following settings on the DUT to include also the source and destination UDP port numbers into the hash function:

```
ethtool -N enp3s0f0 rx-flow-hash udp4 sdfn
ethtool -N enp3s0f1 rx-flow-hash udp4 sdfn
```

We note that they were not used on the Tester computers (n017 and n019), because DPDK uses a poll mode driver, and thus no interrupts are used, when packets are received.

B. Benchmarking IPv4 Kernel Routing

RFC 8219 recommends different frame sizes for testing, and the smallest frame size for IPv4 is 64 bytes. We used only this one, as higher frame sizes require lower frame rates to saturate the 10 Gigabit Ethernet.

1) Throughput Tests

We performed the throughput test also with fixed port numbers as a basis for comparison, and then with pseudorandom source and destination port numbers in the full ranges recommended by RFC 4814. In addition to that, we have also performed some additional, non-standards tests for comparison. All tests were executed 20 times, then median, first percentile and 99-th percentile were calculated. (Of course, the latter two are the same as minimum and maximum, as the number of tests are less than 100.) In addition that, we have also calculated dispersion defined as:

$$dispersion = \frac{99^{th}\ percentile - 1^{st}\ percentile}{median} \cdot 100\% \quad (1)$$

The bidirectional throughout test results of IPv4 Linux kernel routing using a single destination network per direction are shown in Table I. We note that the same frame rates were applied in both directions and thus a commercial Tester would report the double of it, that is, the number of all frames per second forwarded by the DUT. But we kept ourselves to the numbers reported by our bash shell script as throughput and did not double it, that is, our results show the number of frames per second per direction. (We followed

TABLE I
BIDIRECTIONAL THROUGHPUT TEST RESULTS OF IPV4 LINUX KERNEL ROUTING: SINGLE DESTINATION NETWORK (PER DIRECTION)

Throughput (per direction)	fixed ports (for reference)	random ports (both src & dst)	random **src** ports	increasing **src** ports	random **dst** ports	increasing **dst** ports
median (fps)	887,771	3,402,271	3,469,891	3,485,627	3,400,966	3,447,165
1st percentile (fps)	881,834	3,390,562	3,453,124	3,481,931	3,388,670	3,440,778
99th percentile (fps)	891,175	3,406,345	3,476,563	3,492,188	3,406,254	3,456,056
dispersion (%)	1.05	0.46	0.68	0.29	0.52	0.44

TABLE II
BIDIRECTIONAL THROUGHPUT TEST RESULTS OF IPV4 LINUX KERNEL ROUTING: 256 DESTINATION NETWORKS (PER DIRECTION)

Throughput (per direction)	fixed ports (for reference)	random ports (both src & dst)
median (fps)	3,000,693	2,850,648
1st percentile (fps)	2,807,493	2,774,585
99th percentile (fps)	3,272,018	2,945,068
dispersion (%)	15.48	5.98

this approach also in [7], and its rationale is that when the maximum frame rate for the media may be a limiting factor for the throughput, then it is more meaningful to see the frame rate per direction and not the double of it.)

The median throughput measured using random port numbers (3,402,271fps) is about 3.83 times higher than the median throughput measured using fixed port numbers (887,771fps). We are satisfied with this results, because the speed up was nearly linear, when the traffic was distributed among all 8 CPU cores using random port numbers, which is 4 times higher than the 2 CPU cores, which were in use with fixed port numbers. However, we were surprised by the results of the test, when only the source port numbers were varying and the destination port numbers were fixed. How can they be higher than the results of the tests, when all ports are random? Perhaps, the answer is that random port numbers result in only a *roughly but not smoothly uniform distribution* of the frames among the CPU cores. This statement is also supported by the fact that the median throughput using increasing source port numbers (3,485,627fps) is somewhat higher than the median throughput using random source port numbers (3,469,891fps). The difference is even more salient, when the source port numbers are fixed, and only the destination port numbers are varying: the median throughput using increasing destination port numbers (3,447,165fps) is well visibly higher than the median throughput using random destination port numbers (3,400,966fps).

The fact that varying source port numbers result in higher throughput than varying destination port numbers could be attributed to the fact that the [1024, 65535] source port range is wider than the [1, 49151] destination port range. Therefore, the explanation could be that varying source port numbers from a larger range have greater chance to result in a more smoothly uniform distribution of the frames among the CPU cores, than varying destination port numbers from a smaller range. However, currently it is just a hypothesis, which we check in Section IV.D.

As for the dispersion of the results, increasing source port numbers (dispersion is only 0.29%) helped to achieve the most consistent measurement results, which also seems to support that they provide more uniform distribution of the traffic among the CPU cores, than random source port numbers.

The bidirectional throughout test results of IPv4 Linux kernel routing using 256 destination networks per direction are shown in Table II. It is interesting to compare the throughput results using fixed port numbers (in the first column of Table II) with the first two columns of Table I. On the one hand, the median throughput result with 256 destination networks per direction (3,000,693fps) is about 3.38 times higher than the throughput result with a single destination network per direction (887,771fps), because the 256 networks helped to distribute the traffic among the 8 CPU cores. On the other hand, it (3,000,693fps) is less than the median throughput result with a single destination network per direction with random ports (3,402,271fps), because the routing among the twice 256 networks requires more computation than the routing between two networks.

The significant dispersion (15.48%) of the throughput result with 256 destination networks per direction using fixed port numbers could be explained by the fact that the 8-bit field used to express 256 different destination networks was not enough to achieve a smoothly uniform distribution of the traffic among the 8 CPU cores. This explanation is partially supported by the fact that the dispersion is only 5.98% in the next column, where random ports are used. However, in the same time, the median value is decreased to 2,850,648fps. Thus, perhaps higher number of random bits help repeatability, but they still do not guarantee smoothly uniform distribution of the traffic among the CPU cores.

2) Frame Loss Rate Tests

As the same **`siitperf-tp`** program can be used for *frame loss rate* tests (but with a different bash shell script), we did not do any frame loss rate measurements.

3) Latency Tests

RFC 8219 requires to perform throughput tests at the frame rate determined by the throughput test, that is, the median value. The fact that the first percentiles in Table I and Table II are lower than the median values, indicates that some of the measurements produced frame loss at the median rates. It also means that frame loss may happen during the latency measurements, and even the *latency frames* (special tagged frames for latency measurements) may be lost. If a latency frame is lost, then **`siitperf`** reports the highest possible latency value (please refer to our original paper [7] for more information). To mitigate the effect of this phenomenon to the worst case latency results, we used 50,000 latency frames instead of the at least 500 one required by RFC 8219. (Thus, if only a few latency

TABLE III
LATENCY TEST RESULTS OF IPV4 LINUX KERNEL ROUTING: SINGLE DESTINATION NETWORK (PER DIRECTION)

	using fixed ports (for reference), at 887,771fps rate				using random ports (both src & dst) at 3,402,271fps rate			
	Fwd TL	Fwd WCL	Rev TL	Rev WCL	Fwd TL	Fwd WCL	Rev TL	Rev WCL
median (ms)	0.0148	0.0567	0.0144	0.0573	0.0490	0.2228	0.0491	0.2210
1st percentile (ms)	0.0145	0.0495	0.0138	0.0479	0.0479	0.2065	0.0477	0.2022
99th percentile (ms)	0.0173	0.0868	0.0159	0.0709	0.0503	0.2645	0.0503	0.2481

TABLE IV
LATENCY TEST RESULTS OF IPV4 LINUX KERNEL ROUTING: 256 DESTINATION NETWORKS (PER DIRECTION)

	using fixed ports (for reference), at 3,000,693fps rate				using random ports (both src & dst) at 2,850,648fps rate			
	Fwd TL	Fwd WCL	Rev TL	Rev WCL	Fwd TL	Fwd WCL	Rev TL	Rev WCL
median (ms)	0.0322	0.1039	0.0323	0.1042	0.0356	0.1319	0.0357	0.1314
1st percentile (ms)	0.0320	0.1020	0.0322	0.1000	0.0353	0.1265	0.0355	0.1265
99th percentile (ms)	0.0324	0.1091	0.0325	0.1075	0.0359	0.1371	0.0360	0.1388

frames are lost, then their extreme latency values may be omitted, when calculating the worst case latency as 99.9th percentile, see Section 7.2 of RFC 8219.) The duration of the tests was 120s, the sending of the latency frames started at 60s and they were distributed evenly in the second 60s long interval. The tests were executed 20 times.

The results of the latency measurements with a single destination network per direction are shown in Table III. Explanation: TL means *typical latency* and WCL means *worst case latency*. They are given both for the Forward (Fwd) and for the Reverse (Rev) directions, too. The latency results measured at 887,771fps rate using fixed port numbers are given as a basis for comparison. Of course, the latency results measured at 3,402,271fps rate using random port numbers are not directly comparable with them, but the tendency is well visible: all the latency values became higher, yet they are still very low. The increased latency at a 3.83 times higher rate can be explained by the fact that the higher number of frames had to go through the same network interfaces.

The results of the latency measurements with 256 destination networks per direction are shown in Table IV.

TABLE V
PDV TEST RESULTS OF IPV4 LINUX KERNEL ROUTING:
SINGLE DESTINATION NETWORK (PER DIRECTION)

	using fixed ports at 887,771fps rate		using random ports at 3,402,271fps rate	
	Fwd PDV	Rev PDV	Fwd PDV	Rev PDV
median (ms)	0.0731	0.0918	0.3273	0.3273
1st perc. (ms)	0.0426	0.0458	0.2962	0.2915
99th perc. (ms)	0.0850	1.1987	0.4856	0.8735

TABLE VI
PDV TEST RESULTS OF IPV4 LINUX KERNEL ROUTING:
256 DESTINATION NETWORKS (PER DIRECTION)

	using fixed ports at 3,000,693fps rate		using random ports at 2,850,648fps rate	
	Fwd PDV	Rev PDV	Fwd PDV	Rev PDV
median (ms)	0.1086	0.1081	0.1930	0.2011
1st perc. (ms)	0.1019	0.1016	0.1534	0.1522
99th perc. (ms)	0.1156	0.1142	0.2309	0.2294

We do not go into deeper analysis, the results are presented only to demonstrate the operation of the latency measurements with 256 destination networks using random port numbers.

4) PDV Tests

Packet Delay Variation tests were also performed at the rates determined by the throughput tests. Their duration was 60s and they were executed 20 times.

The results of the PDV measurements with a single destination network per direction are shown in Table V. As expected, the values using random ports at the more than 3.4Mfps rate are also higher, but still low.

The results of the PDV measurements with 256 destination networks per direction are shown only for completeness in Table VI.

C. Checking the Performance of Siitperf

Now, we examine the performance cost of the new features. We can be sure that the bottleneck is always the sender function and not the receiver, because we experienced it so before adding the varying port feature, which increased the tasks of sending function and left the receiver untouched. Following the approach of the previous sections of the paper, we used IPv4 traffic for measuring the performance of **`siitperf`**. We note that the generation of IPv6 traffic does not require more computing power from **`siitperf`**, than the generation of IPv4 traffic, because the frames are always pre-generated, and the modification of the pre-generated IPv4 or IPv6 frames in order to use varying port numbers is the same (performed by the very same code lines).

We have performed the self-test of the Tester on n019. The duration of the throughput tests was 60s and the measurements were executed 20 times.

The maximum frame rate achieved by **`siitperf-tp`** with a single destination network per direction is shown in Table VII. The parameters of the six columns are the same as in the case of Table I. It is important that the CPU clock frequency of n019 was set to fixed 2GHz. (Now it is visible that the performance of **`siitperf`** would have been enough to determine the throughput of IPv4 kernel routing without enabling Turbo Mode on n017). As expected, the performance decrease caused by using random port numbers

TABLE VII
MAXIMUM FRAME RATE ACHIEVED BY SIITPERF-TP IN IPV4 TEST FRAME GENERATION: SINGLE DESTINATION NETWORK (PER DIRECTION)

Throughput (per direction)	fixed ports (for reference)	random ports (both src & dst)	random **src** ports	increasing **src** ports	random **dst** ports	increasing **dst** ports
median (fps)	7,077,704	6,327,653	6,649,018	6,894,070	6,694,193	6,921,322
1st percentile (fps)	6,945,860	6,324,217	6,648,428	6,893,232	6,693,327	6,920,804
99th percentile (fps)	7,150,879	6,327,881	6,649,203	6,894,318	6,694,410	6,921,753
dispersion (%)	2.90	0.06	0.01	0.02	0.02	0.01

TABLE VIII
MAXIMUM FRAME RATE ACHIEVED BY SIITPERF-TP IN IPV4 TEST FRAME GENERATION: 256 DESTINATION NETWORKS (PER DIRECTION)

Throughput (per direction)	fixed ports (for reference)	random ports (both src & dst)	random **src** ports	increasing **src** ports	random **dst** ports	increasing **dst** ports
median (fps)	7,002,871	5,278,075	6,379,851	6,585,938	6,380,259	6,588,070
1st percentile (fps)	6,966,302	5,276,365	6,378,905	6,585,443	6,378,905	6,587,689
99th percentile (fps)	7,183,714	5,278,870	6,380,401	6,586,487	6,380,861	6,588,470
dispersion (%)	3.10	0.05	0.02	0.02	0.03	0.01

(both source and destination) is significant compared to the case, when fixed port numbers are used. If only one of the port numbers is random and the other one is fixed, it requires less computing power. And the application of increasing port numbers causes the least performance loss. Interestingly, it seems that the performance penalty of the varying (random or increasing) destination port numbers is somewhat less than that of the varying source port numbers.

The maximum frame rate achieved by **`siitperf-tp`** with 256 destination networks per direction is shown in Table VIII. Not surprisingly, the achieved frame rates are always lower than in the corresponding fields of the previous table.

As for latency tests, we did not perform any performance tests, because the low number of latency frames do not really influence the performance of **`siitperf-lat`**. (It is so for two reasons: the proportion of the latency frames is negligible, and the very same code performs the modification of the latency frames as that of the normal test frames.)

PDV tests require the most manipulations of the fields of the test frames, and we have tested their performance too. As **`siitperf-pdv`** can be used for a special throughput measurement, which we consider important, we used that scenario for testing. The value of the individual frame timeout was set to 10ms. The duration of the tests was 60s, and the measurements were executed 20 times.

The maximum frame rates achieved by **`siitperf-pdv`** with a single destination network per direction is shown in Table IX. As we expected, the usage of two random port numbers has its performance "costs", yet the performance of the Tester still remained high enough.

The maximum frame rate achieved by **`siitperf-pdv`** with 256 destination networks per direction is shown in Table X. Here the performance of the Tester further decreased, when two random port numbers were used, but it still high enough. (It would have been still enough for benchmarking IPv4 Linux kernel routing even if a fixed 2GHz clock signal had been used at the Tester.)

Therefore, we can lay down that implementing varying (random or increasing) port numbers has its performance costs, but the performance of **`siitperf`** is still enough for benchmarking even IPv4 kernel routing. We note that the achievable rates of IPv6 kernel routing are lower than that of IPv4 kernel routing, and the performance of SIIT implementations is even less.

TABLE IX
MAXIMUM FRAME RATE ACHIEVED BY SIITPERF-PDV IN SPECIAL THROUGHPUT TESTS: SINGLE DESTINATION NETWORK (PER DIRECTION)

Throughput (per direction)	fixed ports (for reference)	random ports (both src & dst)
median (fps)	6,811965	5,317,263
1st percentile (fps)	6,749999	5,316,389
99th percentile (fps)	6,820343	5,318,360
dispersion (%)	1.03	0.04

TABLE X
MAXIMUM FRAME RATE ACHIEVED BY SIITPERF-PDV IN SPECIAL THROUGHPUT TESTS: 256 DESTINATION NETWORKS (PER DIRECTION)

Throughput (per direction)	fixed ports (for reference)	random ports (both src & dst)
median (fps)	6,614,810	4,743,239
1st percentile (fps)	6,614,810	4,742,155
99th percentile (fps)	6,615,634	4,743,652
dispersion (%)	0.01	0.03

D. The Effect of Extending the Port Number Ranges

We repeated the bidirectional throughout test of IPv4 Linux kernel routing using a single destination network per direction (the results of which are shown in Table I) in a way that source and destination port number ranges for both directions were set to [0, 65535]. (This time, we omitted the test with fixed port numbers.)

Our aim was twofold:

- to check if extending the source and destination port number ranges to [0, 65535] results in higher throughput or not,
- to test our hypothesis in Section III.B.1 that varying source port numbers resulted in higher throughput than varying destination port numbers, because their range was wider.

TABLE XI
BIDIRECTIONAL THROUGHPUT TEST RESULTS OF IPV4 LINUX KERNEL ROUTING: SINGLE DESTINATION NETWORK (PER DIRECTION), **SOURCE AND DESTINATION PORT NUMBERS ARE FROM THE FULL [0, 65535] RANGE**

Throughput (per direction)	random ports (both src & dst)	random **src** ports	increasing **src** ports	random **dst** ports	increasing **dst** ports
median (fps)	3,417,509	3,488,126	3,502,828	3,418,800	3467227
1st percentile (fps)	3,405,760	3,483,397	3,497,003	3,414,055	3460797
99th percentile (fps)	3,425,783	3,496,105	3,515,626	3,425,785	3472657
dispersion (%)	0.59	0.36	0.53	0.34	0.34

The results are shown in Table XI. On the one hand, the increase of the port number ranges to [0, 65535] resulted in higher throughput in all five measurements compared to the results, when the port numbers ranges recommended by RFC 4814 were used (please refer to Table 1), although the actual increase was marginal, e.g. 0.45% (from 3,402,271fps to 3,417,509fps), when all ports were random. However, on the other hand, the throughput is still higher in the case, when only the source port number is varying then in the case, when only the destination port number is varying, thus our hypothesis in Section III.B.1 is refuted. Even using the same [0, 65535] ranges for source and destination port numbers, the throughput (3,488,126fps) is about 2% higher, if the source ports are random and the destination ports are fixed, than it is (3,418,800 fps) in the case, when the destination port numbers are random and the source numbers are fixed. Finding the root cause of the difference is beyond the scope of our paper, we surmise that perhaps the usage of the source port number and of the destination port number in the hash function is not completely symmetrical.

V. DISCUSSION OF THE RESULTS AND FUTURE WORK

Let us compare the method to use random source and destination port numbers as recommended by RFC 4814 and our original idea to use simply increasing source port numbers. It would be tempting to say that our original idea was better, as it could ensure more smoothly uniform distribution of the traffic among the CPU cores (and thus higher throughput), and it also caused less performance decrease of the Tester. However, this conclusion would be incorrect for at least two reasons:

1. Such a generalization regarding the higher throughput from a single particular case is deliberately unfounded.
2. The aim of benchmarking is not to produce as high as possible results, but to provide realistic performance characteristics.

As for source port numbers in real life traffic, they are very likely better modelled by random port numbers than by one-by-one increasing port numbers. Therefore, source port numbers should be random, even if pseudorandom number generation involves higher computational cost.

As for destination port numbers, we contend that one-by-one increasing destination port numbers are definitely very far from what can be seen in real life traffic. But, we consider uniformly distributed random destination port numbers in the [1, 49151] range also a bad model of reality, because there are a few extremely popular applications like http (port 80) and https (port 443), etc. However, we do not state that it is worth refining the model of the destination port numbers, because of what we mentioned about the exchanging of the roles of source and destination port numbers in Section III.A. This is why **`siitperf`** makes no restriction on the ranges of source and destination port numbers and lets the user to set anything in the [0, 65535] range.

It is beyond the scope of our current paper, but it would be interesting to examine the computational cost of random number generation versus that of rewriting the different fields of the frame.

We consider the performance analysis of stateful NAT64 very important and we plan to create a stateful NAT64 Tester reusing the code base of **`siitperf`**.

VI. CONCLUSION

We have disclosed our design considerations and implementation decisions of enabling **`siitperf`** for using varying port numbers. Our design was flexible enough for supporting both RFC 4814 compliant pseudorandom source and destination port numbers in the ranges specified by the user and different computationally cheaper solutions including that only one of the port numbers is varying and the other one is fixed, as well as one-by-one increasing or decreasing port numbers.

We have examined and demonstrated the how the above mentioned different solutions influence the benchmarking results. We have also measured, to what extent the different solutions decrease the performance of **`siitperf`**.

We conclude that we were successful in implementing the varying port number feature of **`siitperf`**, while keeping its high performance.

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 and Miku Takuma for their help and advice in StarBED usage related issues.

The author thanks Alfred C. Morton for his advice about RFC 4814 and for recommending the setting of the different parameters independently.

The author thanks István Pilisi, National Media and Telecommunications Authority (NMHH), Hungary for the information about the varying port feature of the Spirent SPT-N4U Tester.

The author thanks Keiichi Shima for reading and

commenting the manuscript.

REFERENCES

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

[2] G. Lencse and Y. Kadobayashi, "Comprehensive survey of IPv6 transition technologies: A subjective classification for security analysis", *IEICE Transactions on Communications*, vol. E102-B, no. 10, pp. 2021–2035, DOI:10.1587/transcom.2018EBR0002

[3] C. Bao, X. Li, F. Baker, T. Anderson, and F. Gont, "IP/ICMP translation algorithm", *IETF RFC 7915*, 2016. DOI: 10.17487/RFC7915

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

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

[6] G. Lencse, "Benchmarking stateless NAT64 implementations with a standard tester", *Telecommunication Systems*, DOI: 10.1007/s11235-020-00681-x

[7] G. Lencse, "Design and implementation of a software tester for benchmarking stateless NAT64 gateways", *IEICE Transactions on Communications*, DOI: 10.1587/transcom.2019EBN0010

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

[9] 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

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

[11] G. Lencse, K. Shima, "Performance analysis of SIIT implementations: Testing and improving the methodology", *Computer Communications*, vol. 156, no. 1, pp. 54-67, April 15, 2020, DOI: 10.1016/j.comcom.2020.03.034

[12] G. Lencse and D. Bakai, "Design and implementation of a test program for benchmarking DNS64 servers", *IEICE Transactions on Communications*, vol. E100-B, no. 6. pp. 948–954, Jun. 2017. DOI:10.1587/transcom.2016EBN0007

[13] G. Lencse, "Benchmarking authoritative DNS servers", *IEEE Access*, vol. 8. pp. 130224–130238, Jul. 2020. DOI: 10.1109/ACCESS.2020.3009141

[14] A. C. Morton, "Re: [bmwg] An Upgrade to Benchmarking Methodology for Network Interconnect Devices -- Fwd: New Version Notification for draft-lencse-bmwg-rfc2544-bis-00.txt", May 22, 2020, *IETF BMWG mailing list archive*, [Online]. Available: https://mailarchive.ietf.org/arch/msg/bmwg/xEhrqdP59PAphKJES9viKM8Tt_E/

[15] G. Lencse, Á. Kovács, K. Shima, "Gaming with the Throughput and the Latency Benchmarking Measurement Procedures of RFC 2544", *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, vol 9, no 2, pp. 10-17, 2020, DOI: 10.11601/ijates.v9i2.288

[16] DPDK Documentation, "rte_eth_tx_burst()", [Online]. Available: https://doc.dpdk.org/api/rte__ethdev_8h.html#a83e56cabbd31637efd648e3fc010392b

[17] T. Herbert, W. de Bruijn, "Scaling in the Linux Networking Stack" [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

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

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