# Implementation details to reduce the latency of an SDN Statistical Fingerprint-Based IDS

Alessandro FAUSTO and Mario MARCHESE
*DITEN*
*University of Genoa*
DITEN, Via Opera Pia 11A, Genoa, Italy
alessandro.fausto@unige.it, mario.marchese@unige.it

***Abstract*—The paper represents the first implementation step of a statistical fingerprint based Intrusion Detection System (IDS) exploiting the SDN architecture already in the state of the art. The IDS collects traffic data and implements a suitable machine learning based algorithm to detect the possible presence of malware within the data traffic, implementing the data management scheme within a Ryu SDN controller. The analysis of the performance of the SDN infrastructure by which the Statistical Fingerprint-Based IDS has been implemented identified critical issues. The first issue to tackle is the delay introduced by the SDN hardware/software, which may hinder the practical application of the IDS. This paper presents the improvements applied to the SDN infrastructure in order to minimize the delays introduced by the SDN software infrastructure in a Ethernet-based network, in view of an application over SCADA industrial systems. The analysis focuses on the peak delays that correspond to the action due to the arrival of the first packet of each new flow for which there are not rules in the flow tables of the SDN switch yet. The implemented actions are described in detail. The obtained results are really promising.**

***Index Terms*—Intrusion Detection System (IDS), Software Defined Networking (SDN), Data Plane Development Kit, Ryu, openFlow**

## I. Introduction

Intrusion Detection Systems (IDS) are hardware/software components or groups of devices and components aimed at monitoring a network or a system to detect malicious activity. The paper originally appearing in [1] introduces a statistical analysis based intrusion detection system, which, after extracting a statistical fingerprint, uses a machine learning classifier to decide whether a flow is affected by malware or not. In parallel with the evolution of IDSs, the need of simplifying network management has brought to the development of the Software Defined Networking (SDN) paradigm, which decouples data and control actions. Data forwarding functions are located within devices (switches, routers, gateways) called SDN switches. Control functions are concentrated in SDN controllers whose communication with SDN switches is managed through the OpenFlow signaling protocol. The basic idea in [2], which is an evolution of [1], is implementing the malware detector IDS in [1] within a Ryu SDN architecture. [2] provides the main functional blocks of the proposed architecture also reported in Fig. 1.

This paper tackles the first steps to get a real implementation of the architecture in [2] and presents the results of the measurements about the delay introduced by the SDN statistical IDS infrastructure on network packets. The objective is to verify to what extent it is possible to reduce delays and if they are compatible with the minimum response times of SCADA industrial plants. The paper is structures as follows. The next section describes the used system and its technical details. Section III presents the used testbeds and shows the tackled problems and the results obtained through the actions used to reduce the delay. Section IV contains the conclusions.

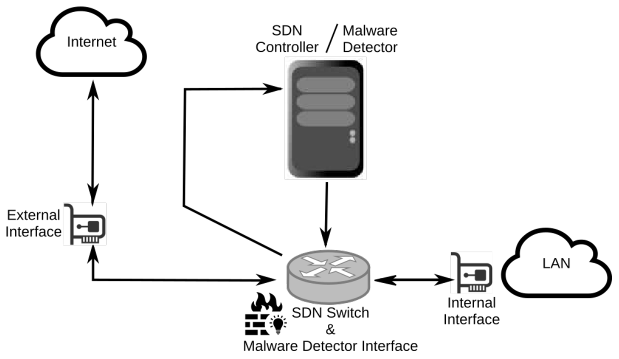

Fig. 1. SDN statistical IDS infrastructure

## II. Description of the implemented SDN IDS infrastructure

The system consists of an SDN switch installed to intercept all the traffic on the local network. The SDN switch is connected through a network interface dedicated to an SDN controller that manages flow tables. The flows are identified through the classic vector composed of UDP/TCP-IP protocol header fields: IP Source and Destination (IP SRC and DST), Source and Destination Ports (SRC and DST Ports), Protocol. When fully operational, the tables contain the management rules for each flow that passes through the switch. Each packet that belongs to a flow not yet present in the tables (in practice the first packet of each new flow) is sent to the SDN controller for analysis. At the end of the analysis the SDN controller adds appropriate rules concerning the flow inside the SDN switch and allows forwarding the flow packets. In more detail: for each packet that belongs to a flow not yet ”cataloged”, three new Openflow communications are generated (Packet-In Message, Modify Flow entry Message and Packet-Out

Message) and the new flow is kept inside the data structure of the IDS code. Instead, for each packet belonging to flows already present in the flow table, the SDN switch simply applies the rules by forwarding or dropping them. In order to minimize the impact of the SDN IDS infrastructure on the data forwarding operation, there are two bottlenecks that prove to be the most relevant. The first one is the time needed to analyze the packet received from the SDN switch and it is closely related to the IDS code structure. The second one is related to the software nature of the SDN infrastructure we built. To carry out a ”trivial” packet forwarding many software components communicating with each other and requiring a lot of lines of code come into play: RX / TX queues of Ethernet drivers, context switches between Kernel and user space, and so on. Furthermore, this problem is amplified by the fact that, for each packet belonging to a new flow, three openFlow messages are exchanged. The SDN-based IDS infrastructure has been implemented through the use of open source software and Linux operating system (OS). The SDN switch is based on Debian or Mint Linux systems with at least three Ethernet network cards (100Mbit or 1Gbit) and Open vSwitch software (OVS) SDN switches. The SDN controller is based on a Gentoo Linux system and, as said, a Ryu SDN software controller where the IDS code is added. Please see sections III-D, III-E and III-F below for more information on software and hardware used. By acting on both software (Linux Kernel, OVS, Ryu) and hardware (CPU, Ethernet card) it is possible to minimize the delays associated with the SDN infrastructure such as the ones about the exchange of openFlow messages between switches and controllers, and the delays inside the SDN switch. Acting on the IDS code, the times required for the analysis of the first packet of each new flow have been minimized.

## III. TestBeds, Implemented Actions, and Results

Testbeds are composed of 2 computers provided with 3 network cards connected each other through two dedicated Ethernet links and connected to the local network through the left network card (Fig. 2). The two computers have different roles: the first one ($A$) acts as SDN controller; the second one ($B$) as SDN switch.

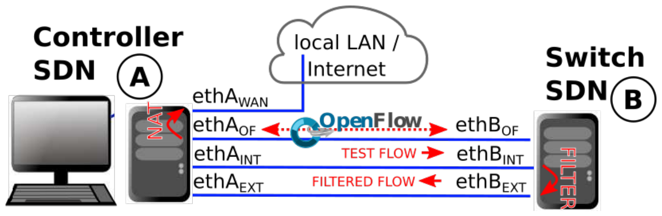

Fig. 2. TestBed structure

$A$ (SDN controller) uses Linux Gentoo OS and Ryu software (dev-python/ryu-4.26) to act as a Controller. It has a CPU quadcore Intel® Core™i5-3340 CPU @ 3.10GHz, 16 GB RAM and has been provided with 4 network cards:

| | |
|---|---|
| $ethA_{WAN}$ | internet/local lan. |
| $ethA_{OF}$ | openflow dedicted connection. |
| $ethA_{INT}$ | network to analize (internal side). |
| $ethA_{EXT}$ | network to analize (external side). |

Three different testbeds have been implemented by only varying $B$ ($B_1, B_2, B_3$) with the aim of obtaining increasingly reduced latencies at the cost of ever-increasing complexity of the software side and increasingly binding hardware demands. All computers $B$ are provided with a Linux Debian or Mint OS and 3 network cards:

| | |
|---|---|
| $ethB_{OF}$ | openflow dedicated connection (internet for updates). |
| $ethB_{INT}$ | network to analyze (internal side). |
| $ethB_{EXT}$ | network to analyze (external side). |

$B1$ is an appliance Nokia IP120 provided with a AMD Geode 266 MHz CPU, 128 MB of RAM, and 3 Ethernet 10/100 Mbit cards. $B_2$ is a PC provided with a quad core Intel Q6600 CPU, 4 GB of RAM, and 3 Ethernet 10/100/1000 network cards. $B_3$ is a PC provided with an Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz, 16 GB of RAM, and 3 Ethernet 10/100/1000 cards.

### A. Network connections

| | |
|---|---|
| $ethA_{WAN}$ | connected to local network and internet. |
| $ethA_{OF}$ | directly connected with $ethB_{OF}$ dedicated to openFlow connection and to possible updates of $B$ |
| $ethA_{INT}$ | directly connected with $ethB_{INT}$ (port #1 switch SDN) and used to simulate the internal network traffic (INT) that $B$ must analyze/filter |
| $ethA_{EXT}$ | directly connected with $ethB_{EXT}$ (port #2 switch SDN) and used to receive the traffic (EXT) that has been analyzed/filtered by $B$. |

### B. Logical infrastructure

Within $B_x$ computers a virtual SDN switch has been created by using an OVS software. It is made of two logical gates (#1 $ethB_{INT}$, #2 $ethB_{EXT}$). The first one assigned to $ethB_{INT}$ and so to input traffic. The second one to $ethB_{EXT}$ and consequently to output (filtered) traffic. The openFlow connection of the control plane (Ryu - Ovs) happens through the dedicated connection $ethA_{OF}$ e $ethB_{OF}$. $A$ acts as SDN controller and receives the first packet of each new flow detected by $B$ over INT or EXT networks. Then it creates and sends the monitoring rules to $B$ which, at each predetermined interval, sends the statistics of all the monitored flows to the controller. At predetermined intervals the controller, through a trained Machine Learning algorithm (ML), catalogs the flows and adds the rules for the management of the flows according to their assignation (normal or malware). For more information see the already mentioned [1] and [2].

### C. Testbed functioning

The entire testbed has been designed to work in real time with constant performance monitoring. $A$ has the possibility to completely control the flow of the INT and EXT networks. In this way it can analyze in detail the results of the filtering action performed by $B$. $A$ is responsible for creating the data flow of the INT network that is filtered in real time by $B$. The data flow of the INT network is created through the port mirroring of the $eth_{WAN}$ connection with the chance to add any network sample (normal or malware) to the traffic. At the same time $A$ receives the network traffic outgoing from the SDN switch through the EXT network and should therefore be cleaned of infected flows. Through the analysis of the flows sent on INT and received by EXT it is possible to verify the correct elimination of the infected flows. It is also possible to measure the delay that each individual packet has suffered from the instant when it has been sent from the interface $ethA_{INT}$ to the instant when it has been received at the interface $ethA_{EXT}$. Automatic scripting engines act on $A$ and measure the OS performance of both computers and individual software OVS and Ryu every 10 seconds. The SDN switch acts as a pass-through filter having the only task of filtering the flows whose transmission statistics are considered by the classification algorithm belonging to a class of malware. In order to measure the minimum performance required by our SDN filtering system, systems $B_1, B_2, B_3$ with increasing performance and complexity have been taken into consideration. The research is currently focused on the use of dedicated computers operating with Linux OS and Open vSwitch software (OVS) excluding (for now) hardware implementations.

### *D. TestBed $A + B_1$*

$B_1$ system has been used to check the minimum limit of computational performance that can be used to correctly perform the filtering operations related to the SDN switch (forwarding of the first packet of each new flow, managing of routing rules and sending statistics related to rules). We have chosen to use a minimal Debian distribution with only the packages necessary to use the OVS software and an access via ssh.

*1) Issues $A + B_1$:* The $B_1$ system did not present any particular overloads during the whole test period and the OS, even with limited resources, has always remained responsive. This testbed has been fundamental to bring out two related problems. Our implementation of the SDN IDS created by relying on the Ryu software introduces delays in the phases both of managing the new flows and of acquiring the statistics aimed at flow classification and filtering (carried out periodically). These delays are introduced for each first packet of each new flow. This is due to the fact that the forwarding rules of the new flow are added by the SDN controller only after the analysis phase of the first packet of that flow is completed. We have empirically seen that the frequency of the new flows detected by the SDN switch (and therefore sent to the controller for analysis) is not constant over time but there is a tendency to create bursts of new flows and consequently bursts of packets sent to the Ryu controller, added to a FIFO processing queue from which they are picked up sequentially. So delays are added together with a consequent rapid extension of the response times as clear in Fig. 3.

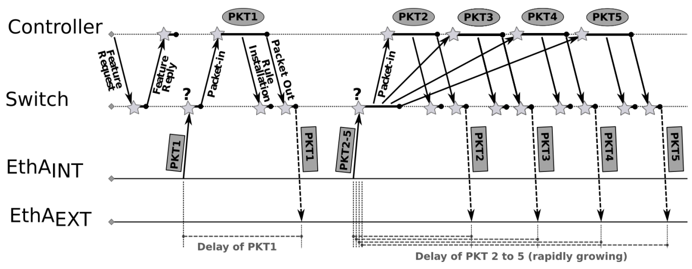

Fig. 3. Increased delay due to burst of new flows

The majority of the bursts of new flows are UDP traffic within the local network. The low percentage of bursts of TCP traffic is linked to the fact that in the traffic of our local network the activity of internet browsing and the consultations of PDF documents, software packages or Linux ISO images that have a payload of consistent size and stable connections is predominant. This feature translates into a low rate of new flows per second. The analysis time of a single new data flow (in Fig. 4) was about 2 ms but, in the worst cases in the presence of bursts, it could even reach 14 ms.

*2) Improvement $A + B_1$:* The code of the demonstrative SDN IDS presented in [2] has been revised and optimized. The supporting data structure used to trace the new data flows has been re-implemented by using a hash table optimized for access through the flow identifier. This action, combined with other minor modifications, has led to a reduction of the analysis time of a single new data flow (in Fig. 5) at 1 ms of average time with peaks of 3 ms.

The second problem is related to the fact that the computation of the statistics and the cataloging action performed by the ML algorithm is carried out within the main process of the Ryu controller which, for this motivation, is no longer able to process other requests (including the reception of new flows) until the end of the cataloging phase. This causes peaks in the queue length of pending requests (most are packetIn) with the relative increase in delay suffered by the packets of the new flows. To solve this problem, the analysis of the statistics of the flows and their classification have been moved to a specific thread that processes them in parallel so that Ryu can proceed with the management of the other requests received by the SDN switch.

We then moved on to measure the delay times caused to the test data flow by the SDN IDS system. To do this, the Tshark software has been used to capture packets simultaneously from the $ethA_{INT}$ and $ethA_{EXT}$ interfaces by printing the MD5 hash of the entire Ethernet package, the reception timestamp, and other information for each packet.

```
tshark -t e -o frame.generate_md5_hash:TRUE -
i enp5s0 -i enp3s0 -Tfields -e frame.md5_hash -
e frame.time_epoch -e frame.number -e ip.src -e
ip.dst -e tcp.srcport -e tcp.dstport -e udp.srcport -e
udp.dstport -Y "not stp and not arp and not loop"
```

These values have been processed by a python script to compute the difference from the timestamps for each pair of identical packets (same MD5 hash of the Ethernet package). To avoid false positives, the packets that cyclically repeat (not stp and not arp and not loop) have been eliminated through a capture filter. The measured delays (in Fig. 6) have an average value of 25 ms but peaks of 380 ms maximum delay in correspondence of new flow bursts.

### *E. TestBed $A + B_2$*

Once optimized the SDN controller code, we have checked how much an increase in system resources could affect the system performance by decreasing the delays. We made a second testbed by replacing the $B_1$ system with the more efficient $B_2$ system. As expected, the delays introduced by the SDN IDS infrastructure have been further reduced through a more rapid management of the flow table linked to the increase in CPU performance and to the possibility of parallel execution between the various software components of the OVS system and the Kernel. The average delay decreases to a value of 1.72 ms and the delay peaks related to the bursts of new flows to 54 ms. The minimum delays are around 0.2 ms.

### *F. TestBed $A + B_3$*

After implementing the described actions linked to the code and the computational power, we have reached an intrinsic limit due to the very nature of the SDN switch software. From an analysis of the state of the art [3] it emerges that the presence of a scheduler that manages the execution of processes in parallel and other strictly technical elements introduce waiting times that translate into delays that cannot be optimized. To overcome this problem the open source world has created the Data Plane Development Kit (DPDK) that can be exploited by OVS to further improve performance. These libraries have been designed to ensure a faster management of packets received from Ethernet cards at the price of a significant increase in complexity. These libraries take advantage of the latest architectural CPU extensions (hugepages, iommu, vt-x, vt-d, etc) and work in close synergy with the Linux Kernel. First of all it is necessary to act on the boot parameters of the Linux Kernel in order to enable the use of hugepages (pages with a size of $>$ 4Kbytes) and to configure the DPDK libraries so that they take advantage of it. Then it is necessary to reserve a part of the system's CPUs for the execution of the DPDK libraries and of the software that use them, prohibiting their use to the Linux scheduler. Only a small set of network cards can be used because optimized drivers are needed [4] [5]. An excellent guideline for the use of DPDK is reported in [6].

The system $B_2$ did not support the mimimum hardware requirements to use DPDK [7], so we have used $B_3$ provided with an Intel 82571EB/82571GB Gigabit ethernet D0/D1 rev 06 (dual port) network card ). Library DPDK version 19.05.0 has been compiled and installed on $B_3$. The configuration has been done coerently to instructions in [6]. OVS version 2.11.1 has been complied to exploit DPDK libraries.

*1) Improvement $A + B_3$:* Test results (in Fig. 6) shows excellent results further reducing delays and bringing the maximum delay in correspondence of new packets bursts around 11.72 ms.

## IV. Conclusions

To achieve a maximum delay around 10 ms was a goal because of the need to check the possibility of reducing the delay times up to the KPIs [8] [9] [10] of a GOOSE network before implementing any SDN IDS filter for this protocol. These KPIs imply device to device transfer times below 20 ms for non tripping and class P2 / P3 messages, and below 100 ms for non tripping and class P1 messages. Still above threshold are the tripping type and P1 class messages KPIs requiring transfer times $< 10$ ms and tripping type and class P2 / 3 messages KPIs requiring transfer times $< 3$ ms.

## V. Future Work

We plan to check the chance to reduce the response times of the current SDN IDS infrastructure by acting on three fronts: looking for higher performance SDN software controller implementations; using the latest generation network cards; decreasing the load of the SDN controller by introducing more SDN controllers that could mitigate the bursts of packets to be cataloged; speeding up the Machine Learning cataloging phase through the use of special hardware accelerators. We will verify the possibility to use SDN controllers made with compiled source code (C or C ++) and we will compare their performance with the Ryu (python) based ones. We are already currently creating a new testBed in which the SDN controller system is equipped with a new IDS version capable of using HW accelerators. We are also currently working on the new NVIDIA Jetson Nano embedded card.

## References

[1] L. Boero, M. Cello, M. Marchese, E. Mariconti, T. Naqash, and S. Zappatore, "Statistical fingerprint-based intrusion detection system (sf-ids)," *International Journal of Communication Systems*, vol. 30, no. 10, 2016.

[2] F. Bigotto, L. Boero, M. Marchese, and S. Zappatore, "Statistical fingerprint-based ids in sdn architecture," in *SummerSim-SPECTS - Society for Modeling & Simulation International (SCS)*, Bordeaux, FR, France, Jul. 2018.

[3] R. Giller. Open vswitch with dpdk overview. [Online]. Available: https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview

[4] Dpdk linux drivers. [Online]. Available: http://doc.dpdk.org/guides/linux_gsg/linux_drivers.html

[5] Dpdk overview of networking drivers. [Online]. Available: http://doc.dpdk.org/guides/nics/overview.html

[6] Configure open vswitch with data plane development kit on ubuntu server 17.04. [Online]. Available: https://software.intel.com/en-us/articles/set-up-open-vswitch-with-dpdk-on-ubuntu-server

[7] Dpdk system requirements. [Online]. Available: http://doc.dpdk.org/guides/linux_gsg/sys_reqs.html

[8] (2016) Substation communication with iec 61850 and application examples. Page 18. [Online]. Available: http://www04.abb.com/global/seitp/seitp202.nsf/0/4d1c836b9e7fdb67c12580870047d7c8/$file/1.Chile_+ABB+_Substatio+communication+with+IEC+61850+and+application+examples.pdf

[9] V. Sushil Joshi, ABB Ltd, "Utilization of goose in mv substation," in *16th national power systems conference*, Hyderabad, IN, India, Dec. 2010, table II. [Online]. Available: http://www.iitk.ac.in/npsc/Papers/NPSC2010/6114.pdf

[10] (2015) Iec 61850 ... the electrical scada standard and integration with ddcmis. Page 37. [Online]. Available: https://nebula.wsimg.com/a49e00efad15d7b63f58b0ff8bd94956?AccessKeyId=1C24E49FE84FF4D32384&disposition=0&alloworigin=1

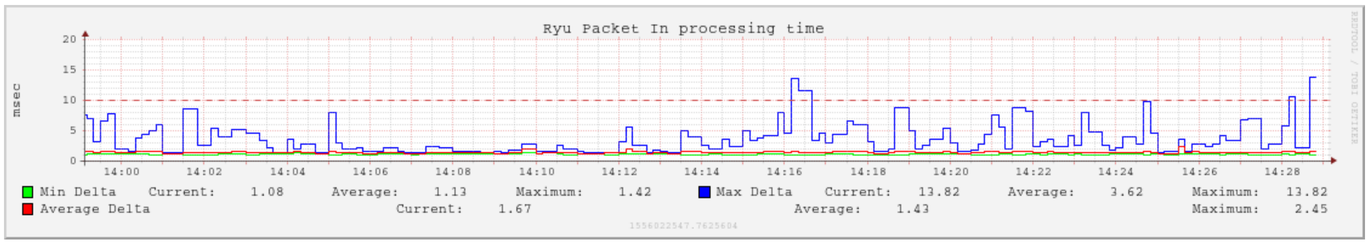

Fig. 4. IDS array structure testBed $A + B_1$

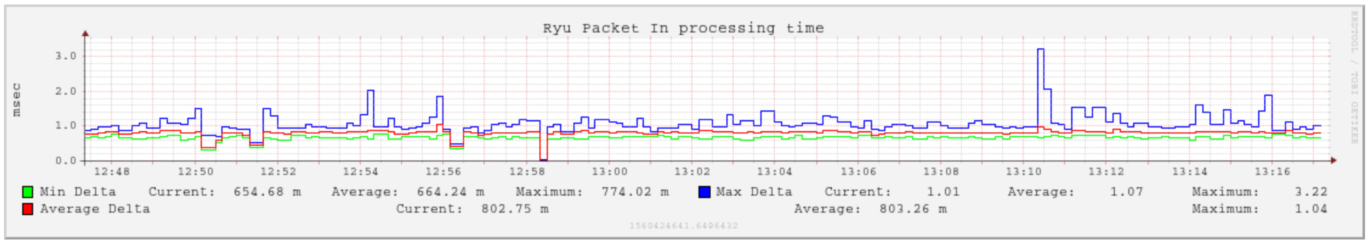

Fig. 5. IDS hash structure testBed $A + B_1$

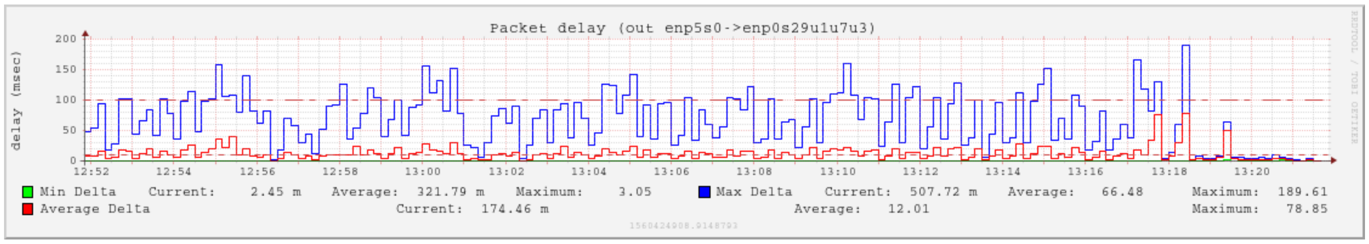

Fig. 6. packet delay testBed $A + B_1$

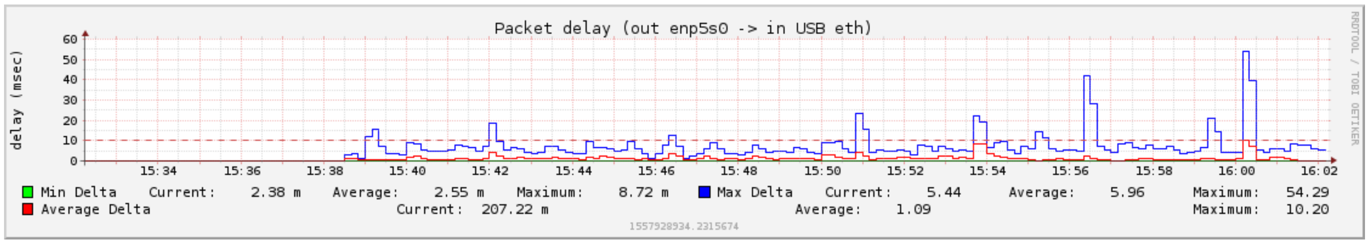

Fig. 7. packet delay testBed $A + B_2$

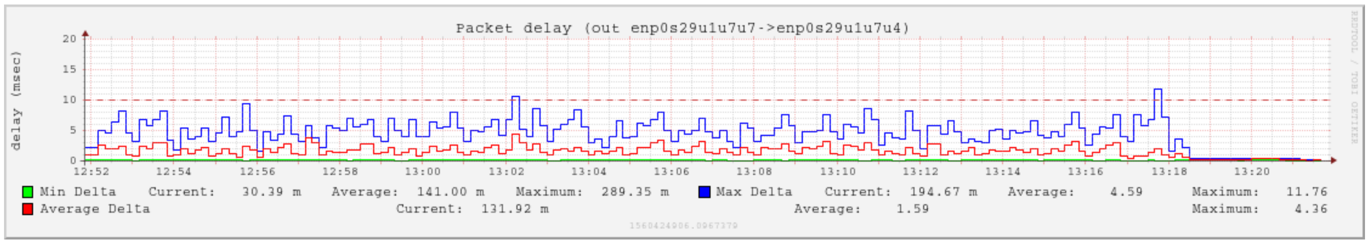

Fig. 8. packet delay testBed $A + B_3$