Sel-INT: A Runtime-Programmable Selective In-Band Network Telemetry System

Shaofei Tang, Deyun Li, Bin Niu, Jianquan Peng, and Zuqing Zhu, *Senior Member, IEEE*

***Abstract*—It is known that by leveraging programmable data plane, in-band network telemetry (INT) can be realized to provide a powerful and promising method to collect realtime network statistics for monitoring and troubleshooting. However, existing INT implementations still exhibit a few drawbacks such as lack of runtime-programmability and relatively high overheads due to per-packet operation. In this work, we propose and design a runtime-programmable selective INT system, namely, Sel-INT, to resolve these issues. Specifically, we first design a runtime-programmable selective INT scheme based on protocol oblivious forwarding (POF), and then prototype our design by extending the famous OpenvSwitch (OVS) platform to obtain a software switch that supports Sel-INT and implementing a Data Analyzer to parse, extract and analyze the INT data. Our implementation of Sel-INT is verified and evaluated in a real network testbed that consists of a few stand-alone software switches. The experimental results demonstrate that Sel-INT can not only adjust the sampling rate of INT in runtime but also program the corresponding data types dynamically, and they also confirm that our proposal can ensure proper accuracy and timeliness for network monitoring while greatly reducing the overheads of INT.**

***Index Terms*—In-band Network Telemetry (INT), Software-defined networking (SDN), Protocol-oblivious forwarding (POF), OpenvSwitch (OVS), Runtime programmability.**

I. Introduction

OVER the past decade, there have been revolutionary changes on and dramatic expansions of the Internet globally. Specifically, the fast developments of datacenters [1, 2] and 5G networks [3, 4] have stimulated various new network infrastructures, such as elastic optical networks (EONs) [5–8], and numerous innovations on networking technologies, *e.g.*, software-defined networking (SDN) [9–11], network virtualization [12–14], and network function virtualization (NFV) [15–17]. Consequently, the Internet is becoming more and more flexible and programmable at the cost of increased complexity. This means that it would be more challenging to troubleshoot the networks and restore them from unclear failures (*e.g.*, congestion, link failure, and black-hole) [18–20]. Therefore, researchers are motivated to develop new network monitoring technologies that can localize the root causes of soft/hard failures quickly without much operational cost or disturbing the ongoing network services.

The traditional network monitoring approaches are usually based on the server-client model. For instance, SNMP [21] lets a network control and management (NC&M) system pull statistics from its network elements for non-realtime data collection, sFlow [22] provides a network-wide view by sampling interface statistics regularly and packet information randomly, and NetFlow [23] collects information regarding IP flows with certain sampling rate and sends the aggregated results to a flow collector for analysis. Although these approaches have been proven to be effective in earlier days of the Internet, their limitations on achieving more sophisticated monitoring are also obvious. Firstly, they need to run an agent on each router/switch for collecting data and responding to polling requests, which costs processing cycles of its local processor and thus might impact the throughput of packet forwarding. Secondly, the bandwidth used for transmitting collected data to the NC&M system would become massive in a large scale network with many routers/switches. Thirdly, they can hardly catch the real-time status of a highly dynamic network. Lastly but most importantly, they cannot reveal the end-to-end information of an arbitrary flow with high accuracy.

The rising of OpenFlow-based SDN [24] has made network monitoring much easier, since the switches in the data plane has to report their feature information to the SDN controller(s) periodically for realizing centralized NC&M. However, some of the old issues, such as relatively long latency in data fetching, incomplete end-to-end per-flow information, and non-scalable processing burdens on switches, still exist and can only be resolved with a more programmable data plane. Meanwhile, people have demonstrated that a programmable and protocol-independent data plane can be achieved by leveraging either the programming protocol-independent packet processor (P4) [25] or the protocol oblivious forwarding (POF) [26, 27]. Therefore, in-band network telemetry (INT) [28] has been developed to provide a network operator the capability of customizing their own network monitoring scheme. Specifically, according to the instructions precoded in the INT headers of a user packet, an INT-capable switch encodes network statistics as specific INT header fields and inserts them in the user packet, for collecting end-to-end per-packet information in real-time. Hence, INT greatly improves the real-time visibility of a network and makes the diagnosis on it much easier.

Previously, there have been a few interesting and powerful P4-based INT implementations on hardware or software platforms [29–31]. For example, developed on a commercial field programmable gate array (FPGA) board, the INT implementation in [29] can collect per-packet information in realtime at 100 Gbps line-rate. However, the hardware implementations are still expensive and inflexible, which makes it difficult for them to be adopted in network environments with virtualization. Although software platforms such as PISCES [32] and

S. Tang, D. Li, B. Niu, J. Peng, and Z. Zhu are with the School of Information Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, P. R. China (email: zqzhu@ieee.org).

Manuscript received on December 28, 2018.

BMv2 [33] can also be leveraged to realize P4-based INT, their packet processing pipelines cannot be adjusted at runtime. This means that a switch has to be taken offline and re-complied every time when we need to change the INT pipeline in it.

The necessity of realizing a runtime-programmable INT system is twofold. Firstly, in order to achieve closed-loop NC&M, we may need to adjust the INT schemes on switches dynamically, *e.g.*, adding new INT pipelines for diagnosing certain flows with a closer look. Secondly but more importantly, sampling network status in a per-packet manner might not be necessary when the line-rate is relatively high and would lead to excessive over-sampling. For instance, at 10 Gbps, the time difference between two adjacent 1024-Byte packet is only 0.8192 μs, and the network usually would not change dramatically in such a short time. Therefore, instead of always relying on per-packet INT, we consider a selective INT scheme as more reasonable. Specifically, the selective INT should be able to adjust the sampling rate of INT in runtime according to the actual needs of network monitoring. Moreover, the locations to collect INT data and the corresponding data types should also be programmable to not only save the packet overhead due to INT but also relieve the processing burdens on switches. Nevertheless, to the best of our knowledge, such a runtime-programmable selective INT scheme has not been designed or demonstrated before.

In this work, we first propose and design a runtime-programmable selective INT scheme based on protocol oblivious forwarding (POF) [34, 35], namely, Sel-INT, and then prototype our design by extending the famous OpenvSwitch (OVS) platform [36]. Our implementation of Sel-INT is verified and evaluated in a real network testbed that consists of a few stand-alone software switches. The experimental results validate that Sel-INT can not only adjust the sampling rate of INT in runtime but also program the corresponding data types dynamically, and they also show that restricting the sampling rate below 20% can greatly improve the packet processing throughput of our software switch while the system's monitoring accuracy on fast-changing INT data is still relatively high even with a sampling rate of 7.1%. Therefore, Sel-INT can ensure proper accuracy and timeliness for network monitoring while greatly reducing the overheads of INT.

Our contributions in this work are summarized as follows:

- We extend OVS to make it POF-enabled, and obtain a brand-new software-based POF switch, namely, OVS-POF, which can process packets in a protocol-agnostic way without severe performance degradation.
- We design and implement Sel-INT over OVS-POF, which is a runtime-programmable network monitoring system that not only achieves much better tradeoff between monitoring accuracy and INT overhead, but also supports changing monitoring schemes (*i.e.*, locations to collect INT data, INT data types, and sampling rate for selective INT insertion) in runtime.
- We design and implement a high-performance data analyzer, which can capture, parse and store the INT data carried by packets, with a packet processing throughput of ~ 2 million packets per second (Mpps).

The rest of the paper is organized as follows. Section II provides a brief survey on the related work. We describe the design of Sel-INT in Section III, while its implementation details are presented in Section IV. The experimental demonstrations and evaluations are discussed in Section V. Finally, Section VI summarizes the paper.

II. Related Work

Before P4-based INT was proposed and demonstrated, people have tried to collect network statistics with other in-band methods. For instance, NetSight was developed in [37] to capture packet processing history and make every processing procedure be visible for helping network diagnosis. Jeyakumar *et al.* [38] proposed the idea of introducing new network monitoring functionality into the data plane with the tiny packet program (TPP), and demonstrated a hardware prototype based on NetFPGA. However, since the TPPs are actually embedded into packets by end-hosts, the scheme might make the network insecure when there are malicious end-hosts. The authors of [39] designed Everflow as a packet-level telemetry system that implements a packet filter together with the "match-and-mirror" functionality in the commodity switches in a datacenter network. Nevertheless, such a match-and-mirror scheme might have difficulty to reveal the end-to-end information of a packet in real-time.

The technical specification of INT has been released in [28], where the authors laid out the overall INT system and provided a few examples on INT use-cases. In [40], in-situ operations, adminstration, and maintenance (IOAM) was specified to record operational and telemetry information in a packet when it transverses an IOAM-domain, and the data fields and associated data types for IOAM were also defined.

P4-based INT has been demonstrated in a Mininet environment for different purposes in [30, 31, 41, 42]. The authors of [30] showed how to debug network by observing HTTP latency instantaneously via INT implemented on P4-based software switches. NetVision was developed in [31], which leverages dynamically-generated double-stack probes instead of normal packets to collect network statistics. However, as the probes might not experience exactly the same network environment as packets, the accuracy and timeliness of monitoring could be affected. Hyun *et al.* [41] studied how to utilize INT to realize knowledge-defined networking, and they prototyped their proposal based on ONOS [43] and BMv2 [33]. More recently, a P4-based selective INT scheme has been proposed in [42]. With a similar idea of TPP [38], the authors realized both the selective INT header insertion and sampling ratio adjustment at the source hosts. Nevertheless, similar as TPP, this scheme might bring in security breach when the end-hosts could not be completely trusted, and it is also not fully runtime-programmable since the the locations to collect INT data and the corresponding data types can hardly be changed in runtime. Moreover, the scheme was only prototyped in Mininet without sophisticated performance optimization.

A few hardware P4-based INT implementations have been discussed in [29, 44–46]. Netcope [29] implemented P4-based INT with an FPGA board that can process packets at a line-rate of 100 Gbps. The project of Deep Insight has been presented

in [44], where P4-based INT is implemented on a high-performance application-specific integrated circuit (ASIC) to make a network be visible for every packet processing in real-time. The authors of [45] considered how to apply INT in a packet-over-optical network and realize multilayer telemetry with an intent-driven framework.

However, as we have explained, P4-based INT schemes have difficulty to adjust the packet processing pipelines at runtime. Moreover, none of these aforementioned studies has demonstrated the selective INT that can program the locations to collect INT data and the corresponding data types and sampling rate, in runtime. This motivates us to study how to design our Sel-INT based on POF [10, 26], which is known to have runtime programmability.

As the major functionality of Sel-INT will be implemented in our home-made software-based POF switch, we summarize the development path of our software-based POF switch as follows. We started the project on software-based POF switch based on the protocol and software architecture discussed in [26]. Then, by leveraging the data plane development kit (DPDK) [47], we accelerated the packet processing, and obtained a software-based POF switch whose packet forwarding throughput is 1 Gbps [10]. Next, we continued to optimize the processing logic in the software switch, and got a better version to report in [35], which achieved a data-rate of 10 Gbps for packet sizes at 512 bytes or longer. However, in the aforementioned studies, we did not consider the well-known open source OpenFlow-based software switch, *i.e.*, OpenvSwitch (OVS) [36]. OVS attracted our interest because of its performance and ecosystem for development. Therefore, in this work, we will first discuss our efforts to add the support of POF in OVS, and then explain how to use the obtained software-based POF switch (*i.e.*, OVS-POF) to implement Sel-INT. To the best of our knowledge, the idea and implementation of supporting POF in OVS have not been discussed in the literature before. Moreover, the performance of OVS-POF is better than the former versions of our software-based POF switches in [10, 35], *i.e.*, it reaches 10 Gbps when the packet size is set as 256 bytes as Section IV-A will show.

III. System Design of Sel-INT

In this section, we lay out the system design of the proposed Sel-INT and define its packet format.

A. System Architecture

Fig. 1 shows the system architecture of our Sel-INT system. Here, to realize runtime-programmability, we design the Sel-INT system based on POF [10, 26], which can also realize a protocol-independent data plane as P4 does. Specifically, POF uses a tuple <*offset*, *length*> to define a packet field, where *offset* refers to the field's start location in a packet and *length* represents its length in bits. Hence, POF switches can locate any field in a packet without referring to a specific protocol, and process packets with pipelines built by the flow tables that are based on the protocol-oblivious forwarding instruction set (POF-FIS) [26]. As the flow tables are actually installed by a POF controller in runtime, the POF switches become runtime-programmable, and image-recompiling is avoided. Previously, we have developed a few network elements [48–51], with which a fully functional POF-enabled network testbed can be built to deliver reasonably good packet processing capacity.

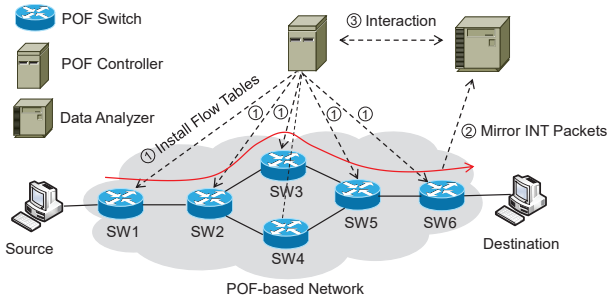


Fig. 1. System architecture of our Sel-INT system based on POF.

Therefore, in Fig. 1, Sel-INT is realized by the POF controller installing INT-related flow tables in the POF switches that are on the forwarding path of a flow. Specifically, when there is a need to apply selective INT on a flow, the controller will first determine the locations to collect INT data, as well as the corresponding data types and sampling rate, then build the flow tables to specify the corresponding INT pipelines, and finally encode the flow tables in *GroupMod* or *FlowMod* messages to install them in the related switches (**Step 1**). The POF controller considered in this work is based on ONOS [43], and we extend its southbound protocol stack to support POF, especially for conveying POF-based group tables.

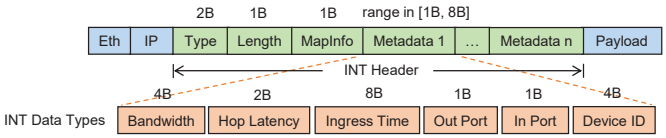


Fig. 2. Format of the INT header used in Sel-INT.

After receiving the flow tables, the switches will build the specified INT pipelines and start to process packets with them. More specifically, the INT pipeline in the ingress switch to the POF-based network is different from those in the subsequent switches on the forwarding path. In the ingress switch (*e.g.*, *SW*1 in Fig. 1), the INT pipeline inserts an INT header into packets (as shown in Fig. 2) selectively, according to the sampling rate predetermined by the controller. Here, the selective insertion is achieved by leveraging the POF-based group tables, and the implementation details will be discussed in Section IV. The INT header in Fig. 2 is inserted after the IP header, where the field lengths are also labeled, *e.g.*, "1B" means a field length of one byte. The *MapInfo* field tells the ingress and subsequent switches about the INT data types to collect and insert into each packet that has the INT header.

***Definition** 1:* In the rest of this paper, we refer to a packet that carries the INT header as an **INT packet**.

Note that, if the controller determines that INT data collection would not be necessary on certain switches on the forwarding path, it would not install the flow table to match

to the INT header on them and thus the switches would process the INT packets and other normal ones in the same way. The next subsection will discuss the details regarding the INT header. Finally, the INT pipeline in the egress switch (*e.g.*, *SW*6 in Fig. 1) duplicates the INT packets and sends them to the data analyzer (**Step 2**). The data analyzer captures the INT packets, parses and extracts the INT data in them, and analyzes the data for realtime network monitoring. Here, to save storage space and avoid unnecessary I/O operations, the data analyzer will filter and hash the extracted data before recording it in a local storage. Meanwhile, the data analyzer interacts with the controller to accomplish closed-loop NC&M (**Step 3**).

B. INT Header

The INT header in Fig. 2 consists of four fields. The first field is *Type*, which has a length of two bytes and is filled with $0x0908$ to indicate an INT header to the POF switches. The one-byte *Length* field follows *Type*, and tells how many *Metadata* fields have already been inserted in the packet for INT. Specifically, each time after a switch finishing an INT operation on the packet, the data in the *Length* field will be incremented by one. We use the *MapInfo* field to indicate the data types to collect on the switches selected for INT. Here, *MapInfo* is essentially a one-byte bitmap, where the lowest six bits correspond to the six types of INT data shown in Fig. 2, respectively, while the highest two bits are reserved for future use. If a switch on the forwarding path is selected for INT, its packet processing pipeline will contain a flow table to match to the INT header and check the *MapInfo* field in it. For all the bits that are turned on, the switch inserts the corresponding types of INT data into each INT packet as new *Metadata* fields. Specifically, each *Metadata* field only contains one type of INT data collected from one switch.

Each INT header can include several *Metadata* fields whose lengths range in $[1, 8]$ bytes. Note that, our Sel-INT supports to insert multiple *Metadata* fields in a packet at one hop (*i.e.*, more than one bits in *MapInfo* gets turned on). The *Metadata* fields defined for the six types of INT data shown in Fig. 2 are explained as follows.

- *Bandwidth*: This INT data uses four bytes to record the average bandwidth usage on the switch port that forwards the packet out. In each POF switch, we assign a counter to calculate the average bandwidth usage of each output port in every 50 msec.
- *Hop Latency*: We use two bytes to record the interval (in μs) between when the packet comes in and when the *Metadata* field gets inserted in it, which is the penultimate action before forwarding it out.
- *Ingress Time*: It has eight bytes to record the local system time (in μs) when the packet comes in.
- *In Port/Out Port*: These two types of INT data use one byte to record the IDs of the input and output ports of the packet, respectively.
- *Device ID*: We use four bytes to record the unique ID of the current switch in the POF-based network. Note that, the ID can be defined in many ways (*e.g.*, using the MAC address of a port on the switch), while in this work, we allocate it manually in sequence.

IV. System Implementation

In this section, we present how to implement our proposed Sel-INT system and show some benchmarking results.

A. OVS Extensions for Supporting POF (OVS-POF)

In order to realize a high-performance POF-enabled software switch that can be utilized to demonstrate the effectiveness of Sel-INT, we decide to extend the famous OVS platform [36] and make it support protocol-oblivious packet forwarding. Our implementation, namely, OVS-POF, is based on OVS v2.6.90 that supports OpenFlow 1.3.

Note that, according to the principle of POF, an arbitrary packet field can be represented by a tuple <*offset*, *length*, *value*>. Hence, we first modify OVS to let it parse each packet field in the form of <*offset*, *length*, *value*> but not according to a specific protocol. This modification ensures that in OVS-POF, both the match fields and the actions' parameter fields are represented and parsed in the form of <*offset*, *length*, *value*>. We also change the protocol stack of the northbound interface in OVS, and make it POF-enabled. Next, we extend OVS to support POF-FIS. Specifically, we redefine the action space in OVS according to POF-FIS and program it to support POF-based actions, such as *add_field*, *modify_field* and *delete_field*, which operate on fields defined as <*offset*, *length*, *value*>.

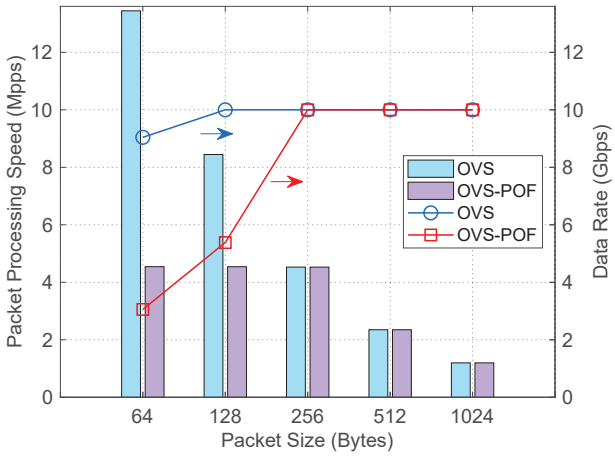


Fig. 3. Fast-path packet forwarding performance of OVS and OVS-POF.

To verify the performance of OVS-POF, we run it on a Linux server that has a 2.10 GHz Intel Xeon CPU and 32 GB DDR3 memory, and benchmark it with OVS. The line-rate of the server's linecard is 10 Gbps, and we compare the fast-path packet forwarding performance of OVS-POF and OVS by changing the packet size from 64 to 1024 bytes. We first run each measurement for a minute to obtain the average results on packet processing speed and data rate, and then for each packet size, the measurement is repeated for three times to average for the final data points. Fig. 3 shows the experimental results. We observe that the packet forwarding performance of OVS-POF is worse than that of OVS for small packets, especially when the packet size is 64 bytes. The reason of this performance degradation is twofold. Firstly, the protocol-oblivious packet forwarding in POF is more flexible and thus more complex

than the protocol-dependent packet forwarding in OpenFlow. Secondly but more importantly, OVS has been optimized for protocol-dependent packet forwarding for a while. Hence, the performance degradation is understandable. Fortunately, the packet forwarding performance of OVS-POF is already good enough for demonstrating the effectiveness of Sel-INT, as we will discuss later in Section V. Meanwhile, we will continue to optimize OVS-POF in our future work.

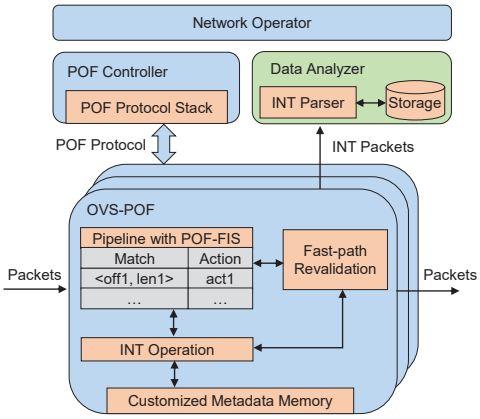


Fig. 4. System implementation of Sel-INT.

B. Implementation of Sel-INT in OVS-POF

Fig. 4 illustrates the detailed implementation of the Sel-INT system. Here, the major part of the implementation is in OVS-POF. More specifically, our work in OVS-POF to support Sel-INT can be summarized as: 1) we extend POF-FIS and design a novel way to realize selective INT header insertion in OVS-POF by leveraging the select group tables (SGTs), 2) we implement an INT operation module in OVS-POF, which can get network statistics from the switch, store them in the switch's customized metadata memory, and provide the required network statistics to be inserted in packets during INT, and 3) we design a fast-path revalidation module to minimize the negative impact of selective INT operations on the fast-path packet processing in OVS-POF, for optimizing the packet processing throughput of the Sel-INT system. In other words, the fast-path revalidation module ensures that the fast-path can be leveraged as much as possible, even though the packets in a flow (*i.e.*, the INT packets and normal ones) can take different actions in a switch.

1) Selective INT Header Insertion based on SGTs: We utilize POF-based SGTs, each of which consists of multiple buckets of actions, to realize selective INT header insertion in ingress switches. Note that, the SGTs in this work operate differently from those defined in OpenFlow. Specifically, OVS selects a bucket from an SGT based on the bucket's weight, and then uses the bucket's actions to process all the packets in a flow (*i.e.*, over the whole service time of a flow, there is no bucket switching). On the other hand, when the flow has a packet coming in, our OVS-POF selects a bucket in the SGT with the probability in proportion to its weight, and then uses the selected bucket's actions to process the packet. Hence, in our design, bucket switching can happen during the flow's service time, to realize selective INT header insertion. Fig. 5 illustrates the SGT designed for Sel-INT.

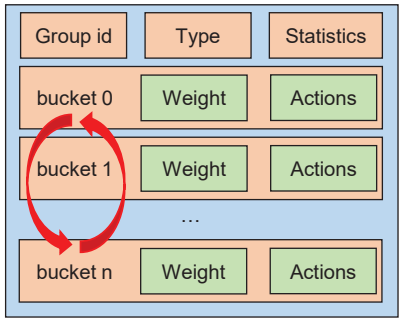


Fig. 5. Design and operation of SGT for Sel-INT.

Algorithm 1: Token-based Bucket Switching

```
for each bucket i in SGT do
    get the bucket's weight and store it in tk[i];
end
while the related flow is active do
    receive a packet;
    flag = 0;
    for i = 0 to len(tk) − 1 do
        if tk[i] > 0 then
            process packet with actions in bucket i;
            flag = 1, tk[i] = tk[i] − 1;
            break;
        end
    end
    if flag = 0 then
        for each bucket i in SGT do
            get the bucket's weight and store it in
            tk[i];
        end
        process packet with actions in bucket 0;
        tk[0] = tk[0] − 1;
    end
end
```

TABLE I
EXAMPLE ON TOKEN-BASED BUCKET SWITCHING

Packet ID	$tk[0]$	$tk[1]$	Selected Bucket
1	2	3	bucket 0
2	1	3	bucket 0
3	0	3	bucket 1
4	0	2	bucket 1
5	0	1	bucket 1

A straightforward way to achieve correct bucket switching based on their weights is to use tokens. If we assume that in each SGT, the buckets' weights are all positive integers, a simple token-based bucket switching algorithm can be designed as in *Algorithm* 1. Specifically, the algorithm traverses the buckets in an SGT in sequence, and use the actions in

a bucket to process packets as long as the bucket's token has not been used up. When the tokens of all the buckets have been used up, their values are restored to the buckets' weights. Table I shows an example on *Algorithm* 1, when there are two buckets in an SGT, and their weights are set as 2 and 3, respectively. Although the token-based bucket switching algorithm is straightforward, the selection of the buckets is unevenly distributed over time, which can degrade the accuracy and timeliness of Sel-INT.

Hence, we consider a polling-based bucket switching algorithm as in *Algorithm* 2 to distribute the selection of the buckets more evenly over time. Here, we first sort the buckets in an SGT in ascending order of their weights (*Line* 1), to avoid unnecessary weight-check operations in *Line* 9. When we try to find the bucket whose weight is the largest in *Line* 9, we always select the one with the highest index if there is a tie among multiple buckets. For the example discussed in Table I, *Algorithm* 2 operates as in Table II and the selection of the buckets distributes more evenly over time. Therefore, we implement *Algorithm* 2 in OVS-POF for bucket switching.

Algorithm 2: Polling-based Bucket Switching

```
1  sort buckets in SGT in ascending order of weights;
2  while the related flow is active do
3      receive a packet;
4      if there is no tk[i] > 0 then
5          for each bucket i in SGT do
6              get the bucket's weight and store it in
               tk[i];
7          end
8      end
9      j = argmax_i(tk[i]);
10     process packet with actions in bucket j;
11     tk[j] = tk[j] − 1;
12 end
```

TABLE II
EXAMPLE ON POLLING-BASED BUCKET SWITCHING

Packet ID	$tk[0]$	$tk[1]$	Selected Bucket
1	2	3	bucket 1
2	2	2	bucket 1
3	2	1	bucket 0
4	1	1	bucket 1
5	1	0	bucket 0

2) Extensions on POF-FIS and INT Pipelines: To ensure that the INT-related actions can be executed efficiently in OVS-POF, we improve the POF protocol to support Sel-INT. Specifically, we extend the *add_field* and *delete_field* actions defined in POF-FIS to obtain two new actions as follows.

- *add_int_field <offset, MapInfo>*: Here, the *offset* indicates the start location to insert INT-related field(s), while depending on the value of *MapInfo*, the action can insert either a new INT header (as shown in Fig. 2) or just a new *Metadata* field. In an ingress switch, the controller will set the *MapInfo* as a valid bitmap according to its definition, *e.g.*, *MapInfo* $= 0x01$ if it wants to monitor "*Device ID*". Then, when OVS-POF sees such a valid bitmap in the *MapInfo*, it inserts a new INT header, which includes the *Type*, *Length* and *MapInfo* fields and the first *Metadata* field, into an INT packet. Otherwise, if the switch is not an ingress one, the controller will set the *MapInfo* as $0xff$, and when OVS-POF sees such an invalid bitmap, it checks the *MapInfo* field in the packet's INT header to know the required type of INT data, and then inserts only a new *Metadata* field accordingly.
- *delete_int_field <offset>*: Here, the *offset* indicates the start location of the INT header to be deleted. As an egress switch, OVS-POF can check the *Length* and *MapInfo* fields in an INT packet to calculate the actual length of its INT header. Hence, no other parameters are required in the *delete_int_field* action.

Fig. 6 illustrates the examples on INT-related group/flow tables on the switches along a forwarding path. The SGT for selective INT header insertion in the ingress switch is in Fig. 6(a), and there are two buckets in it for inserting an INT header and forwarding packet normally, respectively. In *Bucket* 0, the *add_int_field* action inserts a new INT header in the packets of a flow with a sampling rate of $\frac{m}{m+n}$ (*i.e.*, we normally have $m \ll n$). Then, before forwarding the INT packet out, OVS-POF invokes the *modify_field* action to increase the *Length* field in the INT header by one, since a new *Metadata* field has already been inserted. For an intermediate switch that gets involved in Sel-INT, Fig. 6(b) indicates that OVS-POF uses a flow table to match to the *Type* field (*i.e.*, an INT packet has its *Type* field as <*offset* $= 272$ bits, *length* $= 16$ bits> and the value is $= 0x0908$), for detecting INT packets. Next, for each INT packet, it uses the *add_int_field* action to insert a new *Metadata* field, updates the *Length* field in the INT header accordingly, and forwards the packet out. Finally, the egress switch leverages the all group table in Fig. 6(c) to simultaneously perform 1) INT data insertion and INT packet duplication to the Data Analyzer, and 2) INT header deletion and packet delivery to the destination host.

3) Fast-Path Revalidation: To maintain high packet processing throughput, we develop the fast-path of OVS-POF based on that of OVS. However, the logic of OVS' fast-path would prevent bucket switching due to the caching of flow rules, *i.e.*, all the packets in a flow will be processed with the flow rules cached in the fast-path and none of them will be sent to the slow-path to invoke bucket switching there. Specifically, when a packet arrives, OVS first sequentially searches the flow rules for it in the *Microflow* and *Megaflow* caches in its fast-path, and only when the flow rules cannot be found there, it will look up the slow-path and update the fast-path caches. To overcome this issue, we implement a fast-path revalidation scheme in OVS-POF with the operation principle in Fig. 7.

Here, we use the *Revalidator* threads in OVS-POF to delete the flow rules cached in *Microflow* and *Megaflow* periodically and force the packet processing to go through the slow-path and invoke bucket switching there. Specifically, for each flow rule that is related to Sel-INT and gets cached in the fast-path, *Revalidator* sets a threshold on the number of processed packets, and when the threshold is reached, it will delete the flow

Select Group Table	
Bucket 0: Inserting an INT Header	
Weight	Instructions
m	add_int_field<272b, MapInfo> modify_field<288b, 8b, 1> output: next_hop
Bucket 1: Forwarding Packet Normally	
Weight	Instructions
n	output: next_hop

(a) Ingress switch

Flow Table	
Match	Instructions
<272b, 16b>	add_int_field<offset, MapInfo = 0xff> modify_field<288b, 8b, 1> output: next_hop

(b) Intermediate switch

All Group Table
Bucket 0: Mirroring to Data Analyzer
Instructions
add_int_field<offset, MapInfo = 0xff> modify_field<288b, 8b, 1> output: data_analyzer
Bucket 1: Forwarding to End Host
Instructions
del_int_field<272b> output: end_host

(c) Egress switch

Fig. 6. Examples on INT-related group/flow tables on switches along the forwarding path of a flow.

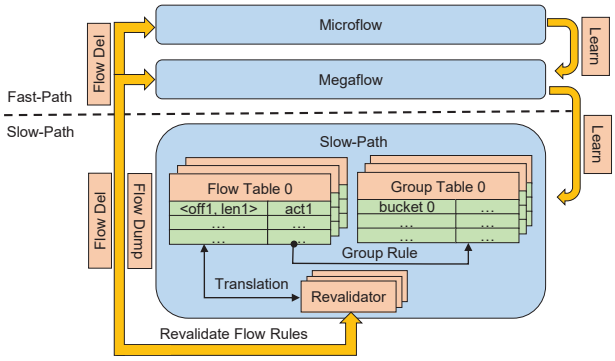


Fig. 7. Operation principle of fast-path revalidation in OVS-POF.

rule from the fast-path caches and force the packet processing go back to the slow-path to check whether a bucket switching should be invoked. Note that, the fast-path revalidation would only affect the packet processing performance of OVS-POF on INT-related flows, and the threshold mentioned above actually determines the period of fast-path revalidation and thus should be optimized. We adjust the threshold to get revalidation periods from 1 to 500 msec, and find that the period of 5 msec is a reasonably good choice, which can not only ensure non-degraded packet processing performance on INT-related flows but also maintain high accuracy and timeliness in INT.

C. Method to Determine Sampling Rate

In our Sel-INT system, the sampling rate of selective INT header insertion is determined and implemented by the SDN controller. Specifically, the SDN controller can analyze historical INT data with the Fourier transform to estimate how fast data samples will vary in the network environment. In the following, we will explain the procedure by using the INT data on bandwidth usage as an example, while the sampling rates of other types of INT data can be determined similarly.

To emulate the traffic fluctuation in a practical network environment, we leverage the real-world traffic trace in [52], scale its bandwidth usage to within $[2, 8]$ Gbps, and sample the trace with an interval of 50 msec[1]. Fig. 8 shows the obtained traffic trace. Then, we apply the fast Fourier transform (FFT) to the traffic trace, and get the single-sided power spectral density (PSD) in Fig. 9(a). We choose the lowest power point in Fig. 9(a), and select its frequency (*i.e.*, $F_N = 7.1$ Hz) as the cut-off frequency, to make sure that the sampling would preserve most of the information in the traffic trace. In other words, the sampling frequency should be $F_S = 2 \cdot F_N = 14.2$ Hz, according to the Nyquist Sampling Theorem. This means that Sel-INT should switch between "inserting an INT header" and "processing packets normally without INT header insertion" for at least 14.2 times per second.

As explained in the previous sub-section, Sel-INT leverages an SGT containing multiple buckets to apply different actions to packets belonging to a same flow. Specifically, by switching among the buckets, the SGT applies the action(s) in selected bucket to packets. In our implementation, an SGT can switch between two buckets for 200 times per second at most. Hence, the analysis above leads to a sampling rate of $14.2/200 = 7.1\%$, which can be realized with an SGT containing two buckets. One bucket is for "inserting an INT header" with weight $m = 1$, and the other one is for "processing packets normally without INT header insertion" with weight $n = 13$.

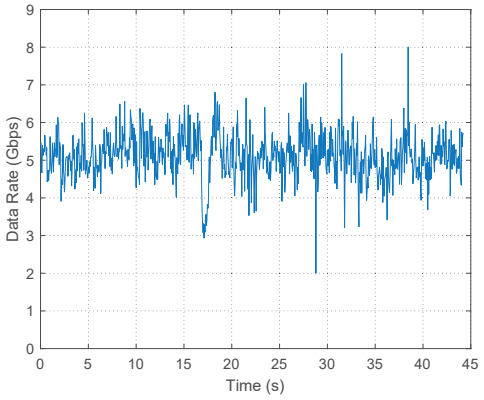
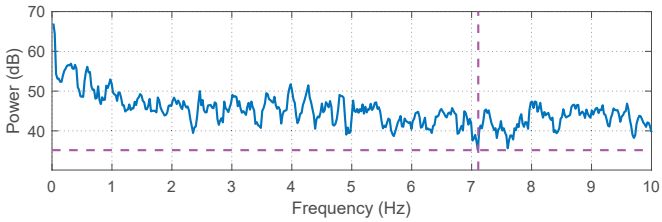


Fig. 8. Traffic trace obtained by scaling the real-world one in [52].

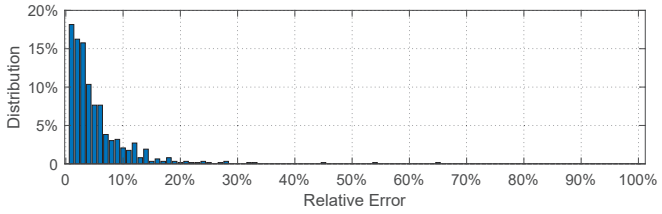
To verify the effectiveness of our approach, we re-sample the modified real-world trace with an interval of 70.4 msec (*i.e.*, 7.1% sampling rate), and get a new trace, namely, *Trace B*. We refer to the original trace in Fig. 8 as *Trace A*. Then, we apply up-sampling and interpolation to both traces and unify their sampling intervals. Next, we compare the two up-sampled traces to get the relative errors caused by the 7.1% sampling rate. The distribution of the relative errors is shown in Fig. 9(b), which indicates that 87.9% of the relative errors

[1]Here, the 50 msec interval is selected because OVS collects the bandwidth usage of a port every 50 msec, and so does our OVS-POF.

are within 10%, while 97.61% of them are within 20%. This confirms that the selected sampling rate ensures relatively high accuracy. In a highly dynamic network environment, analyzing historical INT data helps us get the initial sampling rate to start with. Then, the SDN controller can apply 100% sampling rate over a very short period from time to time to update the historical INT data, and leverage the approach discussed above to adapt the sampling rate to the latest network status.



(a) Single-sided PSD obtained by applying FFT to traffic trace in Fig. 8



(b) Distribution of the relative errors between *Traces* A and B

Fig. 9. Results on obtaining sampling rate of selective INT header insertion.

D. Implementation of Data Analyzer

In our Sel-INT system, an egress switch will duplicate each INT packet to the Data Analyzer, which extracts the INT data in the packet and records the data for network monitoring and further processing. As the processing capacity and storage usage of the Data Analyzer also affect the performance of the Sel-INT system greatly, we design its operation procedure as in *Algorithm* 3. Here, *Lines* 1-6 are for the initialization, and then if the Data Analyzer does receive an INT packet, it will use the for-loop that covers *Lines* 7-33 to process all the *Metadata* fields in the packet's INT header. *Lines* 8 and 9 extract the data in a *Metadata* and hash it to get h, respectively. Next, if the *Metadata* is a new entry, which means that such type of INT data of a flow has not been collected on the location, *Lines* 10-14 create a new entry in the data storage for it. Note that, in our design, each INT data entry in the data storage associates with a counter, which can be updated independently from the records in the entry. Then, if the *Metadata* does not lead to a new entry, *Line* 15 gets the data and hash value in its last record, and the entry's counter is also obtained (*Line* 16).

If the *Metadata* contains fast-changing INT data (*e.g.*, *Bandwidth*, *Hop Latency*, or *Ingress Time*), we calculate the "neighbor difference" between the current and last values as in *Line* 18, and apply a filtering mechanism as in *Line* 19 to avoid recording too many similar records. Here, th_1 and th_2 are preset thresholds, and t and t_r are the current time-stamp and the one of the last record, respectively. We will explain how to determine the values of th_1 and th_2 in the next paragraph. On the other hand, if the *Metadata*'s value only changes slowly over time or actually should not change (*e.g.*, *In/Out Port*, or *Device ID*), the filtering mechanism is designed as in *Line* 26. Specifically, we insert a new record to the *Metadata*'s entry when the filtering condition is satisfied, and otherwise, we only update the counter of its entry.

Algorithm 3: Operation Procedure of Data Analyzer

```
1  while a packet is received do
2      if the packet is not an INT one then
3          continue;
4      end
5      get the current system time as t;
6      parse the packet;
7      for each Metadata field in the INT header do
8          store the Metadata's value in data;
9          h = Hash(data);
10         if the Metadata is a new entry then
11             create a new entry with a counter cnt = 0;
12             store <data, h, cnt, t> as the first record
               in the entry;
13             continue;
14         end
15         get the last record <data_p, h_p, *, *> from the
           Metadata's entry;
16         get the entry's counter to store in cnt;
17         if the Metadata is a fast-changing one then
18             flag = Δ(data_p, data) / min(data_p, data);
19             if flag > th_1 or (t − t_r) > th_2 then
20                 store <data, h, cnt, t> as a new
                   record in the entry;
21                 update the entry's counter as cnt = 0;
22             else
23                 update the entry's counter as
                   cnt = cnt + 1;
24             end
25         else
26             if h ≠ h_p or (t − t_r) > th_2 then
27                 store <data, h, cnt, t> as a new
                   record in the entry;
28                 update the entry's counter as cnt = 0;
29             else
30                 update the entry's counter as
                   cnt = cnt + 1;
31             end
32         end
33     end
34 end
```

The SDN controller determines the value of th_1 by analyzing the distribution of neighbor differences (*i.e.*, the $flag$ in *Line* 18 of *Algorithm* 3). We still use the traffic trace in Fig. 8 as an example to explain the procedure, and the th_1 for other types of INT data can be get similarly. Fig. 10 shows the distribution of the neighbor differences for the trace in Fig. 8, which indicates that 92.07% of the neighbor differences are larger than 1%. Hence, we can empirically set th_1 as

1%. Similar to the sampling rate, the value of th_1 should be updated regularly in a highly dynamic network environment, to adapt to the latest network status. The th_2 is the threshold on the time difference between adjacent records, which avoids not updating the entry of INT data for a relatively long period. Therefore, its value actually depends on the system implementation. In the safest case, we can set th_2 as the shortest interval between which the switches based on OVS-POF update the value of the concerned INT data. For instance, as OVS-POF collects the bandwidth usage of a port every 50 msec, we can just set the th_2 for the INT data regarding bandwidth usage as 50 msec.

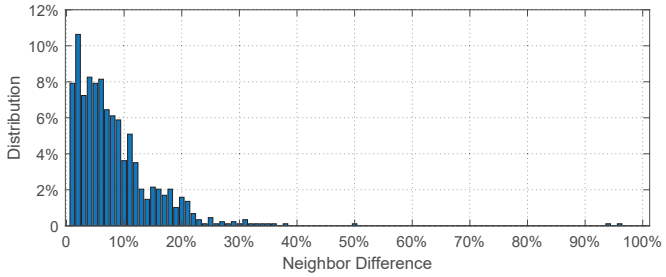


Fig. 10. Distribution of the neighbor differences for traffic trace in Fig. 8.

The operation procedure in *Algorithm* 3 helps us improve the processing capacity and save the storage usage of the Data Analyzer. To verify this, we conduct an INT experiment that only contains an OVS-POF, where the controller asks it to collect all the six types of INT data with 100% sampling. The source host pumps 64-byte packets at 2 million packets per second (Mpps) (*i.e.*, 1.344 Gbps) through the switch for 15 seconds, and we repeat the experiment for three times and average the results to ensure sufficient statistical accuracy. The results in Table III compare the Data Analyzer's performance with and without the filtering mechanism, which suggests that the filtering mechanism improves the processing capacity of the Data Analyzer for more than 37 times, and reduces its memory usage for more than $23,000$ times.

TABLE III
PERFORMANCE BENCHMARKING OF DATA ANALYZER

	Memory usage (MB/s)	Packets processed per second	Packets recorded per second
w/o filtering	369	53,717	53,717
w/ filtering	0.016	1,993,683	20

V. EXPERIMENTAL DEMONSTRATIONS AND EVALUATIONS

In this section, we discuss the experiments to demonstrate and evaluate our proposed Sel-INT system. The experimental setup uses the topology in Fig. 11, which consists of six stand-alone software switches running OVS-POF, an ONOS-based POF controller, a Data Analyzer, and two end hosts. Each OVS-POF runs on a Linux server that has linecards with 10 GbE ports. The traffic generator in the testbed is software-based and realized with the open source software pktgen-dpdk [53], and we run it on a dedicated Linux server. We have calibrated the traffic generator with the procedure in [54], and made sure that it always generates traffic as expected. The traffic analyzer is a hardware-based commercial product. Our experiments include three categories, *i.e.*, feature validation, performance evaluation, and usecase demonstration.

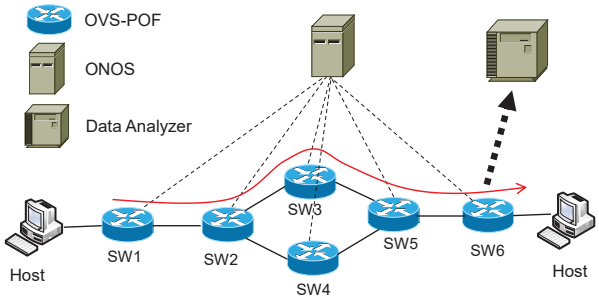


Fig. 11. Experimental setup.

A. Feature Validation

We first perform experiments to verify that the proposed features have been correctly implemented in the Sel-INT system. Specifically, the experiments confirm that our Sel-INT system can achieve selective INT, is runtime-programmable, and can change the locations to collect INT data and the corresponding data types and sampling rate dynamically. Here, we send 64-byte packets at 1 Mpps (*i.e.*, 672 Mbps) to go through the path as indicated by red arrowed line in Fig. 11. During runtime of the network and when the flow is active, we let the POF controller adjust the INT data types to be collected and the corresponding sampling rate, and use the Data Analyzer to receive and parse the INT packets for extracting their *Metadata* fields. More specifically, after Sel-INT has been initialized on all the switches along the path, the runtime adjustments can be easily achieved by the controller updating the Sel-INT policy through sending POF-based *GroupMod* messages to the ingress switch (*i.e.*, *SW*1).

Fig. 12 shows the Wireshark capture of a *GroupMod* received on *SW*1, which includes the instructions regarding the INT data types to be collected and the corresponding sampling rate. Here, we have extended Wireshark to make it POF-compatible. It can be seen that the *GroupMod* contains two buckets, between which the first one (*Bucket* 0) is for normal packet forwarding and has a *weight* of 19, while *Bucket* 1 enables selective INT operations on packets and its *weight* is 1. Hence, the Sel-INT has a sampling rate of 5%. By looking into *Bucket* 1, we can see that the first action in it is *add_int_field*, which tells *SW*1 to insert an INT header into 5% of the packets in the flow, with *offset* $= 272$ bits and *MapInfo* $= 0x31$. Here, the *MapInfo*'s value means that Sel-INT will collect the INT data types of *Device ID*, *Bandwidth* and *Hop Latency*.

Then, we conduct an experiment to demonstrate that the Sel-INT policy can be programmed in runtime. Specifically, the experiment includes four stages: 1) $t \in [0, 26)$ seconds, the controller sets the sampling rate as 10% and *MapInfo* $= 0x01$ (*i.e.*, to collect *Device ID* only), 2) $t \in [26, 53)$ seconds, the controller adjusts the sampling rate to 20% and update *MapInfo* as $0x20$ (*i.e.*, to collect *Bandwidth* only), 3)

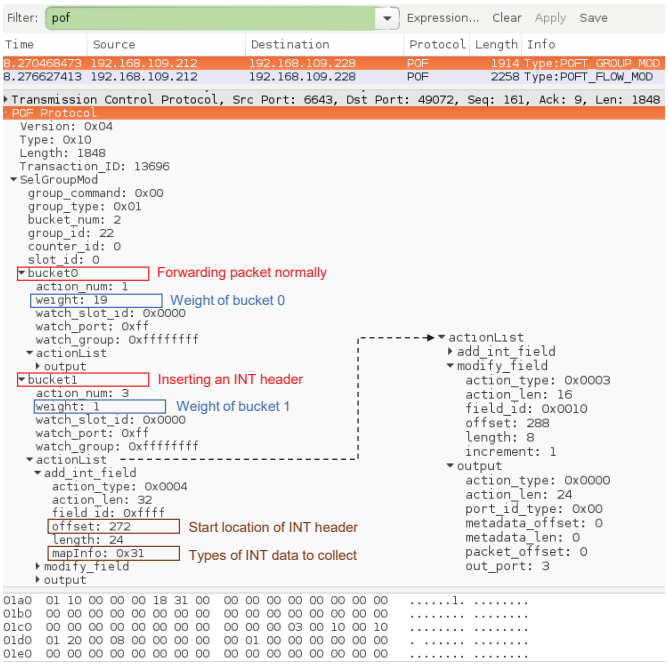


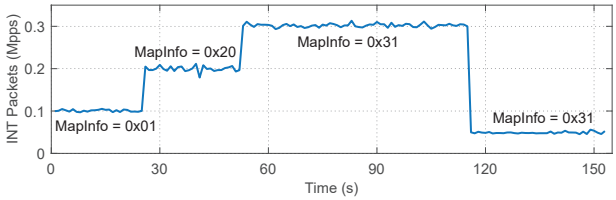
Fig. 12. Wireshark capture of a *GroupMod* received on *SW*1.

$t \in [53, 116)$ seconds, the sampling rate and *MapInfo* get changed to 30% and $0x31$ (*i.e.*, to collect *Device ID*, *Bandwidth* and *Hop Latency*), respectively, and 4) $t \geq 116$ seconds, the controller modifies the sampling rate to 5% and keeps *MapInfo* unchanged. Fig. 13(a) shows how the throughput of the INT packets to the Data Analyzer changes over time. The INT packets' throughput actually reveals the sampling rate since the original throughput of the flow is 1 Mpps. The results in Fig. 13(a) confirm that the sampling rate gets implemented in the Sel-INT system exactly as we designed. Figs. 13(b)-13(d) plot the throughput of the INT packets that contain the INT data related to *Device ID*, *Bandwidth* and *Hop Latency*, respectively, which verify that the types of INT data to collect are programmed correctly too.

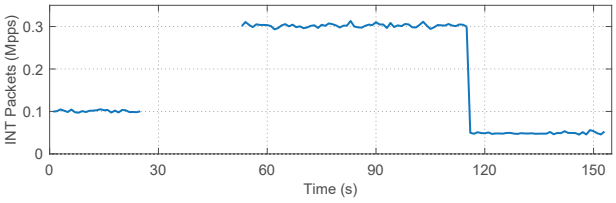
B. Performance Evaluation

1) Packet Processing Throughput during Sel-INT: As Sel-INT needs to invoke bucket switching and involve additional packet processing actions (*e.g.*, *add_int_field* and *delete_int_field*), the packet processing speed of OVS-POF would be affected. Hence, we conduct experiments to measure the packet processing speed of OVS-POF during Sel-INT. Specifically, we install different Sel-INT pipelines in an OVS-POF (*i.e.*, collecting different types of INT data with different sampling rates) and measure its packet processing speed. To consider the worst-case scenario, we pump 64-byte packets through the OVS-POF at the maximum rate that it can handle when no INT is invoked, *i.e.*, 4.538 Mpps.

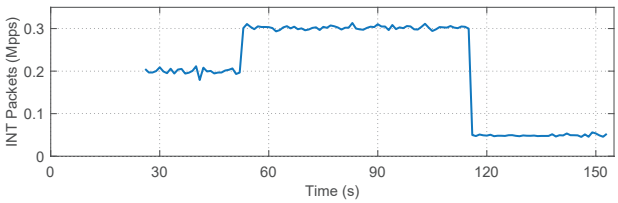
We consider three cases as 1) *MapInfo* $= 0x3f$ to collect all types of INT data, 2) *MapInfo* $= 0x07$ to collect all types of slow-changing INT data (*i.e.*, *In/Out Port* and *Device ID*), and 3) *MapInfo* $= 0x38$ to collect all types of fast-changing INT data (*i.e.*, *Bandwidth*, *Hop Latency* and *Ingress Time*). The



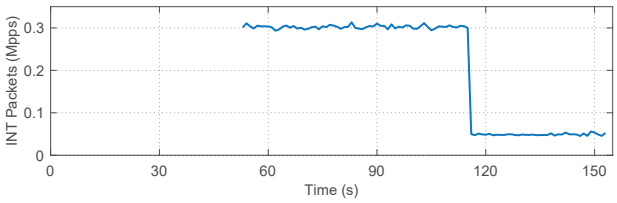
(a) Throughput of INT packets to Data Analyzer in runtime



(b) Throughput of INT packets containing *Device ID*



(c) Throughput of INT packets containing *Bandwidth*



(d) Throughput of INT packets containing *Hop Latency*

Fig. 13. Results on INT packets to Data Analyzer when changing Sel-INT policy in runtime.

experimental results are shown in Fig. 14, and we still obtain them by running each measurement for a minute and repeating each measurement for three times to average for the final data points. It can be seen that for all of the three cases, per-packet based INT (*i.e.*, 100% sampling) would decrease the packet processing speed of OVS-POF significantly. However, if we limit the sampling rate below 20%, the packet processing speed of OVS-POF is 4.029 Mpps in the worst case (*i.e.*, collecting all types of INT data) and the relative degradation is only 11.2%. This verifies the benefit of Sel-INT in saving the processing burdens on switches. Among the three cases, the one that collects all types of INT data provides the lowest packet processing throughput as expected. Meanwhile, it is interesting to notice that the packet processing speed of OVS-POF with *MapInfo* $= 0x07$ is higher than that with *MapInfo* $= 0x38$. This suggests that in Sel-INT, collecting fast-changing INT data is more costly.

2) Benchmarking with Existing Approach: We then use the P4-based selective INT scheme in [42] (*i.e.*, sINT) as the benchmark, which, to the best of our knowledge, is the only existing approach in the literature for selective INT, and conduct experiments to compare our Sel-INT with it. Our Sel-INT is different from sINT from three major perspectives.

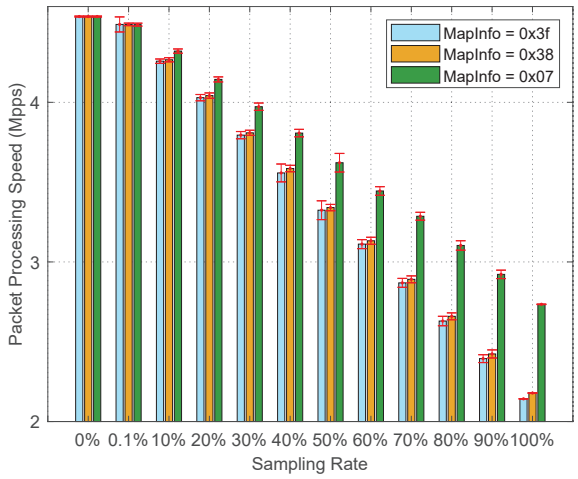
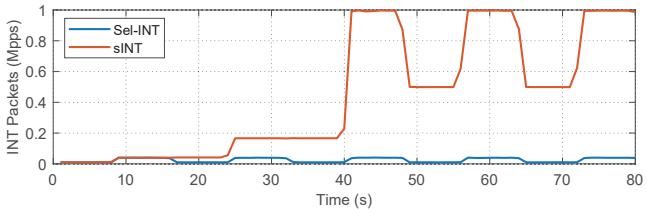


Fig. 14. Packet processing speed during Sel-INT.

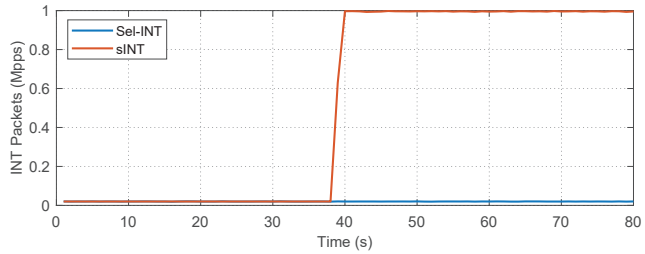
Firstly, since sINT is based on P4-based switches, its packet processing pipelines can hardly be modified in runtime, while our POF-based Sel-INT can change its packet processing pipelines in runtime according to the instructions from an SDN controller. Secondly, sINT adjusts the sampling rate of selective INT header insertion by modifying packet contents at source hosts, while our Sel-INT relies on the SDN controller to instruct switches about sampling rate adjustments. Finally, sINT has to utilize a feedback-based algorithm to adjust the sampling rate. Specifically, it monitors the INT data continuously, if the difference between adjacent data samples goes above a preset threshold, it will double the sampling rate until reaching 100%, and if the data samples do not change significantly over a preset duration (*e.g.*, 10 seconds), it will reduce the sampling rate directly to the minimum value (*i.e.*, 2%). This scheme, however, would make the sampling rate change sharply and unreasonably, which is less effective than the one used by our Sel-INT, *i.e.*, leveraging the global view obtained by the SDN controller to determine the sampling rate.

In order to illustrate the aforementioned differences, we compare sINT and Sel-INT in two experimental scenarios. In each scenario, we conduct the experiment for six runs, and plot the average results in Fig. 15. In *Scenario* 1, we make the packet rate of a flow change according to $[0.5, 1, 0.5, 1, \cdots]$ Mpps, and the flow stays at each rate for 8 seconds, which is slightly shorter than the preset duration in sINT (*i.e.*, 10 seconds). Without a global view on the network, sINT cannot make the duration adaptive. Hence, even if we set its initial sampling rate to the minimum value (*i.e.*, 2%), sINT will double its sampling rate each time when it sees a packet rate change. In consequence, as shown in Fig. 15(a), its sampling rate will quickly increase to 100%, which makes it not "selective" anymore. On the other hand, Sel-INT can recognize the fluctuation of the flow's packet rate precisely and adjust the sampling rate correctly, with the help of the SDN controller. Hence, its sampling rate changes as $[2\%, 4\%, 2\%, 4\%, \cdots]$ in Fig. 15(a), which achieves significant savings on both the INT overhead in the network and the data processing burden in the Data Analyzer. The average standard deviations of the results for Sel-INT and sINT are 0.00116 and 0.00621, respectively.

In *Scenario* 2, we fix the packet rate of a flow at 1 Mpps, but invoke equal-cost multi-path routing (ECMP) in the network. Specifically, the initial sampling rates of sINT and Sel-INT are still set as 2%, but EMCP is invoked at $t = 38$ seconds. With ECMP, the INT data of the flow will indicate two sets of *Device IDs* for the intermediate switches on the flow's routing paths. This, however, will mislead the sampling rate selection algorithm in sINT and make it quickly increase its sampling rate to 100% again, as shown in Fig. 15(b). The average standard deviations of the results for Sel-INT and sINT are 0.00089 and 0.00192, respectively. Since the SDN controller helps Sel-INT detect the situation correctly, its sampling rate does not change throughout the experiment (as illustrated in Fig. 15(b)). The results in Fig. 15 suggest that Sel-INT can determine the sampling rate of selective INT header insertion more accurately than sINT. Meanwhile, we hope to point out that Sel-INT also has other advantages over sINT, *e.g.*, it is totally transparent to end-hosts and thus is securer and more reliable, and with the runtime-programmability enabled by POF, it can change the locations to collect INT data and the corresponding data types in runtime.



(a) Results on throughput of INT packets collected in *Scenario* 1



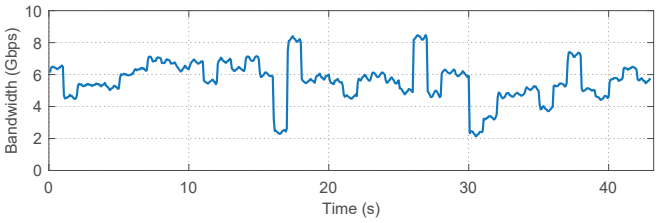
(b) Results on throughput of INT packets collected in *Scenario* 2

Fig. 15. Performance comparisons of Sel-INT and sINT.

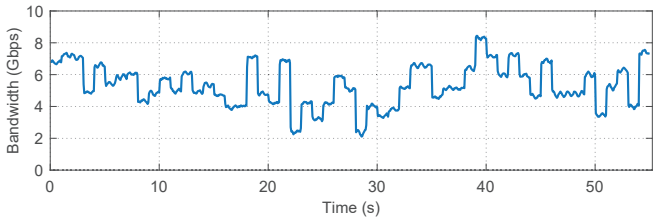
3) Monitoring Accuracy of Sel-INT: In this experiment, we study how the sampling rate and flow pattern affect the monitoring accuracy of Sel-INT. We still make the flow use the forwarding path as indicated by the red arrowed line in Fig. 11. Here, to mimic the traffic fluctuations in a practical network environment, we adopt two real-world traffic traces in [52, 55], refer to them as *Traces* 1 and 2, respectively, and scale their bandwidth usages to within $[2, 8]$ Gbps. The packet size of the flows that use the two traces is set as 1024 bytes, and Sel-INT uses *MapInfo* $= 0x20$ to collect the INT data on *Bandwidth*. We leverage the schemes developed in Section IV to determine the sampling rates of *Traces* 1 and 2 as 7.1% and 8.2%, respectively, and set th_1 and th_2 as 1% and 50 msec, respectively, for both traces. The monitoring results in Figs. 16(a) and 16(b) indicate that Sel-INT can successfully

monitor the bandwidth of highly dynamic flows.

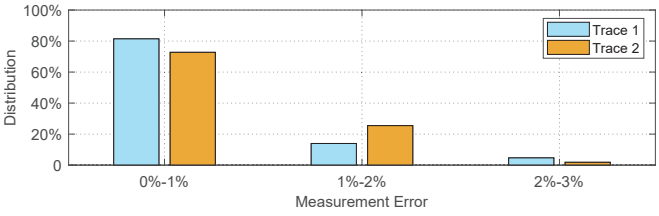
Here, the results are also the average values from six runs for each monitoring experiment, and the average standard deviations of the results to obtain the curves in Figs. 16(a) and 16(b) are 0.15783 and 0.16163, respectively. Note that, with the sampling rates of 7.1% and 8.2%, our Sel-INT system can achieve shorter monitoring intervals than the smallest interval (*i.e.*, 1 second) that our commercial BigTao traffic analyzer can achieve. In order to verify the monitoring accuracy of our Sel-INT system, we average its monitored bandwidth over every second and compare the results with the "ground truth" (*i.e.*, the average bandwidth usage per second in the original traces). Fig. 16(c) shows the distributions of the measurement errors, which indicate that all the measurement errors are below 3%, and for *Traces* 1 and 2, 95.35% and 98.18% of their measurement errors are below 2%, respectively. The results suggest that our proposal can achieve relatively high monitoring accuracy on fast-changing INT data such as *Bandwidth*.



(a) Monitored bandwidth from Sel-INT with 7.1% sampling rate for *Trace* 1



(b) Monitored bandwidth from Sel-INT with 8.2% sampling rate for *Trace* 2



(c) Distribution of measurement errors related to the ground truth

Fig. 16. Results on bandwidth monitoring with Sel-INT.

C. Usecase Demonstration: Path Validation

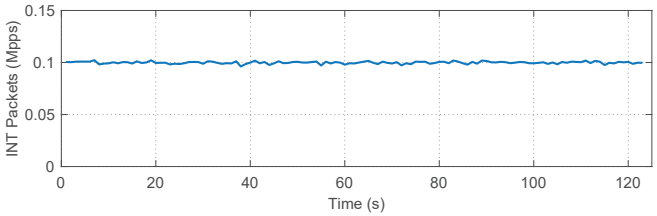
In a network, the forwarding path of a flow can experience some unexpected changes due to either misconfigurations (*e.g.*, incorrect load-balancing and errors in group tables) or intentional attacks (*e.g.*, eavesdropping and man-in-the-middle attack). Path validation with INT can provide a powerful and illustrative way to detect the changes, and our Sel-INT can make this task much easier with more flexibility and significantly less overheads. In the following, we perform experiments to demonstrate the effectiveness of our Sel-INT system using path validation as a usecase. We still send 64-byte packets at 1 Mpps through the path as indicated by red arrowed line in Fig. 11, *i.e.*, *SW*1→*SW*2→*SW*3→*SW*5→*SW*6.

We first consider the scenario in which a misconfiguration on *SW*2 invokes unwanted load-balancing to switch the flow between two paths: *SW*1→*SW*2→ *SW*3→*SW*5→*SW*6 and *SW*1→*SW*2→*SW*4→*SW*5→*SW*6. Meanwhile, the Sel-INT system collects *Device ID* on each hop with a sampling rate of 10% for path validation. The results from the Data Analyzer, which are the average throughputs of the INT packets that contain the *Device IDs* of different switches, are plotted in Fig. 17. It can be seen that the throughputs of the INT packets containing *SW*1, *SW*2, *SW*5 and *SW*6 are normal as expected. However, the INT packets that contain *SW*3 only have a throughput as half of the expect value, while there are unexpected INT packets containing *SW*4. Hence, with the monitoring results in Fig. 17, the network operator can easily figure out that there is a misconfiguration on *SW*2.

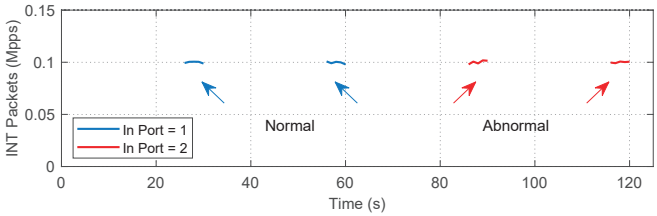
Fig. 17. INT packet throughputs for path validation (misconfiguration).

Next, we emulate the scenario in which an attacker hacks into *SW*2 and *SW*4 to intentionally diverge the flow's forwarding path to *SW*1→*SW*2→*SW*4→*SW*5→*SW*6, for realizing a harmful purpose (*e.g.*, eavesdropping or man-in-the-middle attack). Note that, as the attacker can modify the *Device ID* of *SW*4 and make it impersonate *SW*3, the path validation scheme discussed above would not be able to detect the path change. This can be verified with the results in Fig. 18(a), which shows that the throughput of the INT packets containing *SW*3 stays at the expected value throughout the whole experiment, even though the path has already been changed at $t = 70$ seconds.

Hence, to detect such intentional path changes, the operator needs to collect not only *Device ID* but also *In Port* on each hop along the path. Then, as long as not all the switches have been hacked, the path change can still be detected by looking into the *In Ports* provided by the intact switch(es) (*e.g.*, *SW*5 in this experiment). However, such a monitoring scheme would cause too much overhead. Hence, we let the Sel-INT system invoke 10% selective collection on *In Port* for 5 seconds in every 30 seconds, to reduce the monitoring overhead. Fig. 18(b) shows the monitoring results for *SW*5, which indicate that the flow's input port to *SW*5 has been changed before the third monitoring period on *In Port*. Then, by combining the results in Fig. 18, the operator can infer that certain switches might have been hacked. This further verifies the advantages brought by the runtime-programmability. The results in Figs. 18(a) and 18(b) are also obtained by averaging the results from six runs. The average standard deviation of the results in Fig. 18(a) is 0.00253, while those of the four curves in Fig. 18(b) are 0.00231, 0.00249, 0.00298 and 0.00218, respectively.



(a) Throughput of INT packets containing the *Device ID* of *SW*3



(b) Throughput of INT packets containing the *In Port* at *SW*5

Fig. 18. INT packet throughputs for path validation (intentional attack).

VI. Conclusion

We proposed and implemented Sel-INT as a runtime-programmable selective INT system. Our implementation was verified and evaluated in a real network testbed that consists of a few stand-alone software switches. The experimental results verified that Sel-INT can not only adjust the sampling rate of INT in runtime but also program the corresponding data types dynamically, and they also demonstrated that Sel-INT can ensure proper accuracy and timeliness for network monitoring while greatly reducing the overheads of INT. More specifically, restricting the sampling rate below 20% could greatly improve the packet processing throughput of our software switch, while the system's monitoring accuracy on fast-changing INT data was still relatively high even with a sampling rate of 7.1%. Moreover, we utilized path validation as a simple usecase to show the effectiveness of our Sel-INT system on revealing both unintentional misconfigurations and intentional attacks.

Acknowledgments

This work was supported in part by the NSFC projects 61871357, 61771445 and 61701472, CAS Key Project (QYZDY-SSW-JSC003), and SPR Program of CAS (XDC02010703).

References

[1] P. Lu *et al.*, "Highly-efficient data migration and backup for Big Data applications in elastic optical inter-datacenter networks," *IEEE Netw.*, vol. 29, pp. 36–42, Sept./Oct. 2015.

[2] W. Fang *et al.*, "Joint defragmentation of optical spectrum and IT resources in elastic optical datacenter interconnections," *J. Opt. Commun. Netw.*, vol. 7, pp. 314–324, Mar. 2015.

[3] P. Marsch *et al.*, "5G radio access network architecture: Design guidelines and key considerations," *IEEE Commun. Mag.*, vol. 54, pp. 24–32, Nov. 2016.

[4] L. Liang, W. Lu, M. Tornatore, and Z. Zhu, "Game-assisted distributed decision-making to build virtual TDM-PONs in C-RANs adaptively," *J. Opt. Commun. Netw.*, vol. 9, pp. 546–554, Jul. 2017.

[5] Z. Zhu, W. Lu, L. Zhang, and N. Ansari, "Dynamic service provisioning in elastic optical networks with hybrid single-/multi-path routing," *J. Lightw. Technol.*, vol. 31, pp. 15–22, Jan. 2013.

[6] L. Gong *et al.*, "Efficient resource allocation for all-optical multicasting over spectrum-sliced elastic optical networks," *J. Opt. Commun. Netw.*, vol. 5, pp. 836–847, Aug. 2013.

[7] Y. Yin *et al.*, "Spectral and spatial 2D fragmentation-aware routing and spectrum assignment algorithms in elastic optical networks," *J. Opt. Commun. Netw.*, vol. 5, pp. A100–A106, Oct. 2013.

[8] M. Zhang, C. You, H. Jiang, and Z. Zhu, "Dynamic and adaptive bandwidth defragmentation in spectrum-sliced elastic optical networks with time-varying traffic," *J. Lightw. Technol.*, vol. 32, pp. 1014–1023, Mar. 2014.

[9] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: an intellectual history of programmable networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–98, Apr. 2014.

[10] S. Li *et al.*, "Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability," *IEEE Netw.*, vol. 31, pp. 12–20, Mar./Apr. 2017.

[11] N. Xue *et al.*, "Demonstration of OpenFlow-controlled network orchestration for adaptive SVC video manycast," *IEEE Trans. Multimedia*, vol. 17, pp. 1617–1629, Sept. 2015.

[12] L. Gong, Y. Wen, Z. Zhu, and T. Lee, "Toward profit-seeking virtual network embedding algorithm via global resource capacity," in *Proc. of INFOCOM 2014*, pp. 1–9, Apr. 2014.

[13] H. Jiang, Y. Wang, L. Gong, and Z. Zhu, "Availability-aware survivable virtual network embedding (A-SVNE) in optical datacenter networks," *J. Opt. Commun. Netw.*, vol. 7, pp. 1160–1171, Dec. 2015.

[14] L. Gong and Z. Zhu, "Virtual optical network embedding (VONE) over elastic optical networks," *J. Lightw. Technol.*, vol. 32, pp. 450–460, Feb. 2014.

[15] M. Zeng, W. Fang, and Z. Zhu, "Orchestrating tree-type VNF forwarding graphs in inter-DC elastic optical networks," *J. Lightw. Technol.*, vol. 34, pp. 3330–3341, Jul. 2016.

[16] W. Fang *et al.*, "Joint spectrum and IT resource allocation for efficient vNF service chaining in inter-datacenter elastic optical networks," *IEEE Commun. Lett.*, vol. 20, pp. 1539–1542, Aug. 2016.

[17] Y. Wang, P. Lu, W. Lu, and Z. Zhu, "Cost-efficient virtual network function graph (vNFG) provisioning in multidomain elastic optical networks," *J. Lightw. Technol.*, vol. 35, pp. 2712–2723, Jul. 2017.

[18] R. Govindan *et al.*, "Evolve or die: High-availability design principles drawn from Google's network infrastructure," in *Proc. of ACM SIGCOMM 2016*, pp. 58–72, Aug. 2016.

[19] S. Liu, W. Lu, and Z. Zhu, "On the cross-layer orchestration to address IP router outages with cost-efficient multilayer restoration in IP-over-EONs," *J. Opt. Commun. Netw.*, vol. 10, pp. A122–A132, Jan. 2018.

[20] Z. Zhu *et al.*, "Build to tenants' requirements: On-demand application-driven vSD-EON slicing," *J. Opt. Commun. Netw.*, vol. 10, pp. A206–A215, Feb. 2018.

[21] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A simple network management protocol (SNMP)," *RFC 1098*, May 1990. [Online]. Available: https://tools.ietf.org/html/rfc1157

[22] P. Phaal, S. Panchen, and N. McKee, "InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks," *RFC 3176*, Sept. 2001. [Online]. Available: https://tools.ietf.org/html/rfc3176

[23] B. Claise, "Cisco systems NetFlow services export version 9," *RFC 3954*, Oct. 2004. [Online]. Available: https://tools.ietf.org/html/rfc3954

[24] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[25] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[26] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. of ACM HotSDN 2013*, pp. 127–132, Aug. 2013.

[27] S. Li *et al.*, "Improving SDN scalability with protocol-oblivious source routing: A system-level study," *IEEE Trans. Netw. Serv. Manag.*, vol. 15, pp. 275–288, Mar. 2018.

[28] C. Kim *et al.*, "In-band network telemetry (INT)," *Tech. Spec.*, Jun. 2016. [Online]. Available: https://p4.org/assets/INT-current-spec.pdf

[29] 100G in-band network telemetry with Netcope P4. [Online]. Available: https://www.netcope.com/Netcope/media/content/100G-In-band-Network-Telemetry-With-Netcope-P4.pdf

[30] C. Kim *et al.*, "In-band network telemetry via programmable dataplanes," in *Proc. of ACM SIGCOMM 2015*, pp. 1–2, Aug. 2015.

[31] Z. Liu *et al.*, "NetVision: Towards network telemetry as a service," in *Proc. of ICNP 2018*, pp. 1–2, Sept. 2018.

[32] M. Shahbaz *et al.*, "PISCES: A programmable, protocol-independent software switch," in *Proc. of ACM SIGCOMM 2016*, pp. 525–538, Aug. 2016.

[33] P4 software switch - Behavioral Model version 2 (Bmv2). [Online]. Available: https://github.com/p4lang/behavioral-model

[34] D. Hu *et al.*, "Flexible flow converging: A systematic case study on forwarding plane programmability of protocol-oblivious forwarding (POF)," *IEEE Access*, vol. 4, pp. 4707–4719, 2016.

[35] Q. Sun, Y. Xue, S. Li, and Z. Zhu, "Design and demonstration of high-throughput protocol oblivious packet forwarding to support software-defined vehicular networks," *IEEE Access*, vol. 5, pp. 24 004–24 011, 2017.

[36] B. Pfaff *et al.*, "The design and implementation of Open vSwitch," in *Proc of USENIX NSDI 2015*, pp. 117–130, May 2015.

[37] N. Handigol *et al.*, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. of NSDI 2014*, pp. 71–85, Apr. 2014.

[38] V. Jeyakumar *et al.*, "Millions of little minions: Using packets for low latency network programming and visibility," in *Proc. of ACM SIGCOMM 2014*, pp. 3–14, Aug. 2014.

[39] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," in *Proc. of ACM SIGCOMM 2015*, pp. 479–491, Aug. 2015.

[40] F. Brockners *et al.*, "Data fields for in-situ OAM," *IETF Draft*, Oct. 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ippm-ioam-data-04

[41] J. Hyun, N. Tu, and J. Hong, "Towards knowledge-defined networking using in-band network telemetry," in *Proc. of NOMS 2018*, pp. 1–7, Apr. 2018.

[42] Y. Kim, D. Suh, and S. Pack, "Selective in-band network telemetry for overhead reduction," in *Proc. of CloudNet 2018*, pp. 1–3, Oct. 2018.

[43] ONOS. [Online]. Available: https://onosproject.org/

[44] Barefoot deep insight. [Online]. Available: https://www.barefootnetworks.com/products/brief-deep-insight/

[45] M. Anand, R. Subrahmaniam, and R. Valiveti, "POINT: An intent-driven framework for integrated packet-optical in-band network telemetry," in *Proc. of ICC 2018*, pp. 1–6, May 2018.

[46] B. Niu *et al.*, "Visualize your IP-over-optical network in realtime: A P4-based flexible multilayer in-band network telemetry (ML-INT) system," *IEEE Access*, vol. 7, pp. 82 413–82 423, 2019.

[47] DPDK: Data Plane Development Kit. [Online]. Available: https://www.dpdk.org/

[48] S. Li *et al.*, "SR-PVX: A source routing based network virtualization hypervisor to enable POF-FIS programmability in vSDNs," *IEEE Access*, vol. 5, pp. 7659–7666, 2017.

[49] H. Huang *et al.*, "Realizing highly-available, scalable and protocol-independent vSDN slicing with a distributed network hypervisor system," *IEEE Access*, vol. 6, pp. 13 513–13 522, 2018.

[50] K. Han *et al.*, "Application-driven end-to-end slicing: When wireless network virtualization orchestrates with NFV-based mobile edge computing," *IEEE Access*, vol. 6, pp. 26 567–26 577, 2018.

[51] S. Zhao, D. Li, K. Han, and Z. Zhu, "Proactive and hitless vSDN reconfiguration to balance substrate TCAM utilization: From algorithm design to system prototype," *IEEE Trans. Netw. Serv. Manag.*, vol. 16, pp. 647–660, Jun. 2019.

[52] Caida datasets. [Online]. Available: https://data.caida.org/datasets/passive-2016/equinix-chicago/20160406-130000.UTC/equinix-chicago.dirA.20160406-125912.UTC.anon.pcap.gz

[53] pktgen-dpdk. [Online]. Available: https://git.dpdk.org/apps/pktgen-dpdk/

[54] A. Botta, A. Dainotti, and A. Pescape, "Do you trust your software-based traffic generator?" *IEEE Commun. Mag.*, vol. 48, pp. 158–165, Sept. 2010.

[55] Caida datasets. [Online]. Available: https://data.caida.org/datasets/passive-2016/equinix-chicago/20160406-130000.UTC/equinix-chicago.dirB.20160406-131500.UTC.anon.pcap.gz