Incremental Deployment of Programmable Switches for Network-wide Heavy-hitter Detection

Damu Ding[1,2], Marco Savi[1], Gianni Antichi[3], and Domenico Siracusa[1]

[1]Fondazione Bruno Kessler, CREATE-NET Research Center, Trento, Italy
[2]University of Bologna, Bologna, Italy
[3]Queen Mary University of London, London, United Kingdom
{*ding, m.savi, dsiracusa*}*@fbk.eu*, *g.antichi@qmul.ac.uk*

***Abstract*—The advent of Software-Defined Networking with OpenFlow first, and subsequently the emergence of programmable data planes, has boosted lot of research around many networking aspects: monitoring, security, traffic engineering. In the context of network monitoring, most of the proposed solutions show the benefits of data plane programmability by simplifying the complexity of the network with a one big-switch abstraction. Only few papers look at network-wide solutions, but consider the network as non heterogeneous: only composed by programmable devices. In this paper, we argue that the primary challenge for a successful adoption of those solutions is the deployment problem: how to compose and monitor a network consisting of both legacy and programmable switches? We propose an approach for incrementally deploy programmable devices in an ISP network with the goal of monitoring as many distinct network flows as possible. While assessing the benefits of our solution, we realized that proposed network-wide monitoring algorithms might not be optimized for a partial deployment scenario. We then also developed a novel strategy capable of detecting network-wide heavy flows with the same accuracy of state-of-the-art solutions but by relying on less information from the data plane.**

***Index Terms*—Network monitoring, Programmable data plane, Incremental deployment, Heavy-hitter detection**

I. Introduction

Network monitoring is of primary importance: it is the main enabler of various network management tasks, ranging from accounting [1][2], traffic engineering [3][4], anomaly detection [5][6], Distributed Denial-of-Service (DDoS), and scans detection [7][8]. With the advent of Software-Defined Networking (SDN), the significance of network monitoring has been certainly boosted. This is because, SDN, with the idea of a (logically) centralized control, allows an easy coupling of network management operations with the observed network status. As a result, SDN has been seen as the answer to many of the limitations of legacy network infrastructures [9][10][11]. However, such a noble intent has been limited by its current predominant realisation, the OpenFlow (OF) protocol. Indeed, current OpenFlow APIs are ill-suited and cannot provide accurate data-plane measurements: the main mechanism exposes the per-port and per-flow counters available in the switches [12]. An application running on top of the controller can periodically poll each counter using the standard OF APIs and then react accordingly, instantiating the appropriate rule changes. As a consequence, OF suffers from two important limitations: *(i)* the controller needs to know in advance which flows have to be monitored in the data plane and *(ii)* as the data plane exposes just simple counters, the controller needs to do all the processing to determine the network state.

978-1-5386-9376-6/19/$31.00 ©2019 IEEE

Lately, the advent of the so-called *programmable switches* (e.g. P4-enabled switches [13]) has introduced the possibility to program data plane with advance functionality and enabled the possibility to implement more refined monitoring solutions directly in the switch hardware. Such a disruptive technology has attracted a growing number of researchers and practitioners that in turn have proposed many different solutions to enhance SDN capabilities in the context of network monitoring [14][15][16][17][18]. As a result, the prospect of realizing fine-grained network-wide monitoring, by analyzing the expose information from all the switches in a network, has attracted lot of interest [16][19][20]. However, in practice, a one-shot replacement of all the existing legacy devices with programmable switches is not a feasible solution due to operational and budget burdens. Clearly, this limits the benefits in terms of network flow monitoring performance, since a partial deployment leads to a reduced flow visibility.

This paper proposes a novel approach for an *incremental deployment* of programmable switches in Internet Service Provider (ISP) networks, with the goal of optimizing network-wide monitoring practices. To achieve it, it is important to have visibility over the largest number of distinct flows. To this end, we exploit the HyperLogLog algorithm [21] that is generally used for the count-distinct problem, approximating the number of distinct elements in a multi-set. While assessing the benefits of our solution, we shortly realized that state-of-the-art network-wide monitoring algorithms might not perform optimally in a partial deployment scenario. We therefore propose a new algorithm that is capable of detecting *network-wide* heavy flows (i.e., *heavy hitters*) using as input only partial information from the data plane. We evaluate our incremental deployment strategy alongside the proposed heavy-hitter detection algorithm in simulation. By comparing our solution with the state-of-the-art proposals, the results show that we can achieve, according to the F1 score, a better accuracy using less switches.

The main contributions of the paper are as follows:

- We tackle the problem of partial deployment of programmable networks in the context of network monitor-

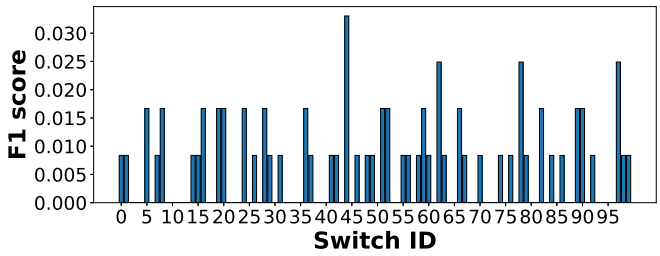


Fig. 1. F1 score of Harrison's heavy-hitter detection strategy with single programmable-switch deployment

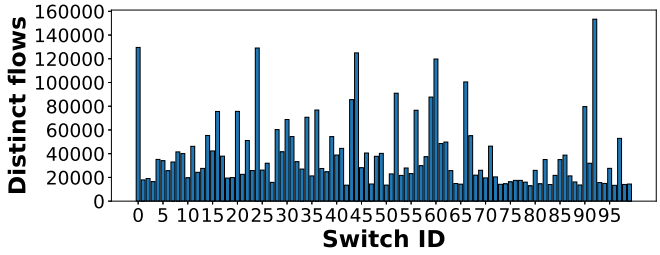


Fig. 2. Number of distinct flows crossing each switch

ing. We propose a new strategy that allows to incrementally deploy new programmable switches and simultaneously maximize the network monitoring operation.
- We propose a new strategy for network-wide heavy-hitter detection which is robust to partial deployments.

The remainder of the paper is organized as follows. In Section II we explain the best practices for an effective partial deployment of programmable switches, while Section III presents the algorithmic background. Section IV describes our incremental deployment algorithm, and Section V presents our network-wide heavy-hitter detection strategy. Section VI reports our simulation results and comparison with the state of the art. Finally, Section VII recalls the related work and Section VIII concludes this paper.

II. Hints for improved monitoring performance with limited flow visibility

When only a limited number of programmable switches can be deployed in a legacy network, the network operator must ensure that they are deployed in such a way it is made the best use of them in terms of monitoring performance, measured by F1 score. Fig. 1 shows the results of a simple test: we simulated a topology of 100 nodes with real traffic, and we evaluated the F1 score of an existing threshold-based network-wide heavy-hitter detection strategy, proposed by Harrison et al. [20] (see Section VI for more details on simulation settings and evaluated metrics) when only one legacy switch/router[1] is replaced with a programmable switch. The graph shows all the 100 possible deployments. What we can see is that the F1 score *(i)* is in all cases low but *(ii)* substantially varies depending on the placement position of the programmable switch. Consideration *(i)* comes from the fact that by replacing only one switch the flow visibility is very low, since only heavy hitters crossing such switch can be detected, while consideration *(ii)* proves that how we deploy programmable switches in the network is a fundamental aspect to ensure good monitoring performance with limited flow visibility.

Our intuition is that, when deploying a single programmable switch, *an effective strategy is to replace the one crossed by the highest number of distinct flows*. This because the highest the number of monitored flows is, the highest (in average) the chance of monitoring some heavy hitters is. Fig. 2 shows the number of distinct flows crossing each one of the switches in the given time interval. Two observations can be made: *(i)* if a switch crossed by a few number of distinct flows is replaced, it is highly probable that it cannot detect any heavy hitter (i.e., F1 score is often zero); *(ii)* a (weak) correlation between F1 score and number of monitored distinct flows indeed exists. For example, replacing the switch with ID 44 leads to the highest F1 score, and the same switch is one of the switches crossed by the highest number of distinct flows. However, the network-wide heavy-hitter detection strategy proposed in [20] has not been explicitly designed to best exploit the available information on switches' monitored distinct flows, unlike our proposed strategy (see Section V), so correlation between F1 score and number of distinct flows is minimal.

The same considerations can be made when more than one programmable switches have to be deployed in the network. In this case, it must be ensured that *the subset of programmable switches to be deployed monitors the highest number of distinct flows overall* (i.e., neglecting duplications): this guarantees satisfactory performance in the execution of monitoring tasks such as heavy-hitter detection. The considerations and intuitions discussed in this section have thus guided us in the design of our algorithm for incremental deployment of programmable switches, which is shown in Section IV.

III. Background

In this section, we present the background for our incremental deployment algorithm and network-wide heavy-hitter detection strategy. Our algorithms rely on different data streaming concepts and methods.

A. Count of distinct flows

An efficient and effective method to count a number of distinct items from a set is HyperLogLog [21]. In our specific case, given a packet stream $S = \{a_1, a_2, ..., a_m\}$, where each packet is characterized by a specific `(srcIP,dstIP)` pair (generically called *flow key*), it returns an estimation of cardinality of flows, i.e., how many `(srcIP,dstIP)` distinct pairs exist in the stream[2]. In this paper, we use $\hat{n} \leftarrow Hll(S)$ as notation to indicate input and output of the HyperLogLog algorithm: Hll indicates the algorithm, S the input packet stream and $\hat{n}$ the cardinality of flows (i.e., number of distinct flows). The relative error of HyperLogLog is only

[1]In the remainder of this paper, we will use the generic term *legacy device* to generically refer to legacy switches or routers. In fact, programmable switches can support both Layer-2 and Layer-3 functionalities.

[2]Note that in this paper, without any loss of generality, we consider source/destination pairs as flow identifiers. However, other definitions could also be adopted (e.g. 5-tuple).

$\frac{1.04}{\sqrt{n}}$ where n is the number of registers. Apart from its high accuracy, HyperLogLog is also very fast since the query time complexity is O(1). Moreover, calculating the *union* (or merge) of two or more HyperLogLog data structures is also very efficient, and can be used to count the number of distinct flows in the union of two (or more) data streams, e.g. S_a and S_b. In our notation, this can be written as $\hat{n}_{union} \leftarrow Hll(S_a \cup S_b)$, where $\hat{n}_{union}$ is the number of distinct flows of the packet streams union $S_a \cup S_b$.

B. Estimation of flow packet counts for heavy-hitter detection

Estimating the number of packets for a specific flow is fundamental for a proper detection of heavy hitters. Different algorithms exist to perform such an estimation: we choose to use Count-Min Sketch [22], which relies on a probabilistic data structure based on pairwise-independent hash functions. To formalize the problem, we consider a stream of packets $S = \{a_1, a_2, ..., a_m\}$. The Count-Min Sketch algorithm returns an estimator of packet count $\hat{f}_x$ of flow $x \sim$ (srcIP$_x$, dstIP$_x$) satisfying the following condition: $Pr[|\hat{f}_x - f_x| > \varepsilon|S|] \leq \delta$, where ε $(0 \leq \varepsilon \leq 1)$ is the relative biased value and δ $(0 \leq \delta \leq 1)$ is the error probability. In Count-Min Sketch, the space complexity is O($\varepsilon^{-1} log_2(\delta^{-1})$), and per-update time is O($log_2(\delta^{-1})$) [23]. Additionally, the estimation of packet count satisfies $f_x \leq \hat{f}_x \leq f_x + \varepsilon|S|$, where f_x is the real packet count value. The accuracy of Count-Min Sketch depends on ε and δ, which can be tuned by respectively defining *(i)* the output size N_h of each hash function and *(ii)* the number N_s of hash functions of the data structure.

In previous definitions, a heavy hitter is a flow whose packet count overcomes a threshold $\vartheta|S|$ $(0 < \vartheta < 1)$. If a Count-Min Sketch is adopted, the probability to erroneously detect a heavy hitter due to packet miscount is defined in the following way: $Pr[\exists x | \hat{f}_x \geq (\vartheta + \varepsilon)|S|] \leq \delta$. If N_s is large enough, error probability δ is negligible.

IV. An algorithm for incremental deployment of programmable switches

In this section we propose a novel algorithm for the incremental upgrade of a legacy infrastructure with programmable switches, which aims at ensuring high network monitoring performance, as discussed in Section II.

A. Problem definition

Our problem of incremental deployment of programmable switches can be formalized in the following way.
Given:

- A network topology of a legacy network infrastructure $\mathcal{G} = (\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ is the set of legacy devices and $\mathcal{L}$ the set of interconnection links;
- A long-term estimation of the transmitted packets in the network between different sources and destinations (i.e., traffic matrix T), including their routing paths and possible re-routing paths in case of failures. From this information it is possible to retrieve the estimated packet stream T_i for each switch $i \in \mathcal{N}$;
- A number $P \leq |\mathcal{N}|$ of legacy devices to be replaced with programmable switches;

Replace a subset of $\mathcal{P}$ (such that $P = |\mathcal{P}|$) of legacy devices with programmable switches with the goal of monitoring the highest number of distinct flows in the network and in an *incremental way*. This means that it must be assured that any subset of programmable switches $\mathcal{Z}$ (with $|\mathcal{Z}| \leq P$) that have been already deployed in the network as intermediate step, monitors the highest number of distinct flows as well.

Algorithm 1: Incremental deployment algorithm

Input: Long-term traffic statistics T, Network topology $\mathcal{G}$, Number of legacy devices P to be replaced
Output: Set of legacy devices $\mathcal{P}$ to be replaced

```
1  max ← 0;
2  P ← {};
3  n̂ ← 0;
4  n̂^pre ← 0;
5  T^pre ← {};
6  key ← empty;
7  for Each legacy device i ∈ N carrying traffic T_i do
8      n̂ ← Hll (T_i);
9      if n̂ > max then
10         max ← n̂
11         key ← i
12 P.add(key)
13 if P > 1 then
14     n̂^pre ← max
15     T^pre ← T_key
16     while P.size() ≤ P do
17         for Each switch i ∈ N \ P carrying traffic T_i do
18             n̂ ← Hll(T^pre ∪ T_i)
19             if n̂ > max then
20                 max ← n̂
21                 key ← i
22         P.add(key)
23         n̂^pre ← max
24         T^pre ← T^pre ∪ T_key
25 return P
```

B. Incremental deployment algorithm

As we mentioned above, HyperLogLog has good performance on estimating the distinct flows from an union of packet streams, so we use it to estimate the number of distinct flows passing through a set of legacy devices. The pseudo code of our proposed algorithm is shown in Algorithm 1. To place the first programmable switch, we compute the estimated number of distinct flows $\hat{n}$ carried by each of the $i \in \mathcal{N}$ legacy devices using the HyperLogLog algorithm. The input of HyperLogLog, for each device $i \in \mathcal{N}$, is T_i. For the replacement with a programmable switch, the algorithm selects the legacy device crossed by the highest number (max) of distinct flows (Lines 7-11). Such legacy device is added to $\mathcal{P}$.

Once the first legacy device has been replaced, the principle to replace any other legacy device consists in progressively finding the one that, if replaced, allows to overall monitor the highest number of distinct flows in the network. To do so, we exploit the *union* property of HyperLogLog (Line 18), recalled in Section III. As shown in Lines 13-24, the algorithm estimates the number of monitored distinct flows $\hat{n}^{pre}$ coming from the union of packet streams *(i)* of all the previously-upgraded programmable switches (T^{pre}) and *(ii)* of any legacy device $i \in \mathcal{N} \setminus \mathcal{P}$ still in the network (T_i). Then, the algorithm selects for replacement (and thus addition to set $\mathcal{P}$) the legacy device i leading to the largest number of monitored distinct flows overall (max). This operation is iterated until a number $P = |\mathcal{P}|$ of legacy switches has been replaced.

Once the set $\mathcal{P}$ has been defined, the network operator can proceed with the physical replacement of the legacy devices with programmable switches, while ensuring interoperability in a hybrid environment [24]. Note that our incremental deployment algorithm focuses on the replacement of switches instead of new additions to the network. In fact, adding new switches may imply changes to existing routing paths and alteration of flow statistics, making our algorithm ineffective.

V. A Network-Wide Heavy-Hitter Detection Strategy Robust to Partial Deployment

Figure 3 shows the interaction between switches and a centralized controller for network-wide heavy-hitter detection, in the case of partial deployment of programmable switches (i.e., when only some switches can perform monitoring operations in the data plane). Time is divided in intervals, and in every time interval each programmable switch dynamically stores in a *sample list* only those flows whose packet count is larger than a dynamic *sampling threshold*. At the end of each time interval, if any programmable switch stores in its sample list one or more flows with packet count larger than a dynamic *local threshold*, it reports a *true flag* to the centralized controller. The flows whose packet count overcomes the local threshold are called *potential heavy hitters*. The controller, if at least one true-flag report is received, then polls all the programmable switches in the network to gather the {flow key, packet count} pairs of all the flows stored in their sample lists. This information is used to estimate the whole *network volume*, i.e., the number of all the unique packets transmitted in the network in the given time interval. Finally, the controller computes the *global threshold* leveraging the estimated network volume and retrieves all the network-wide heavy hitters, i.e., the flows from sample lists whose packet count is larger than the global threshold. Finally, all the flow and packet statistics are reset and a new time interval is started.

In the following subsections, we formalize the problem and describe in detail the algorithms running both in the programmable switches and in the controller to implement the proposed high-level strategy.

A. Problem definition

We formulate the network-wide heavy-hitter detection problem as follows.

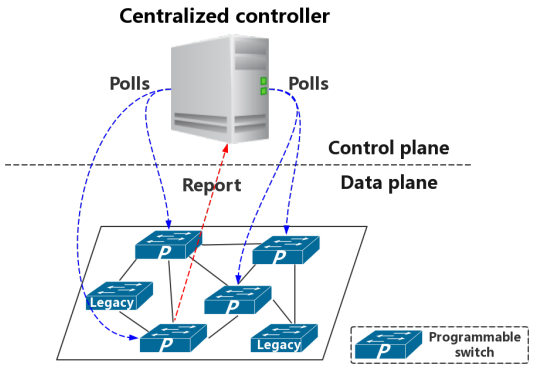


Fig. 3. Interaction between controller and programmable switches for network-wide heavy-hitter detection in a partial deployment scenario

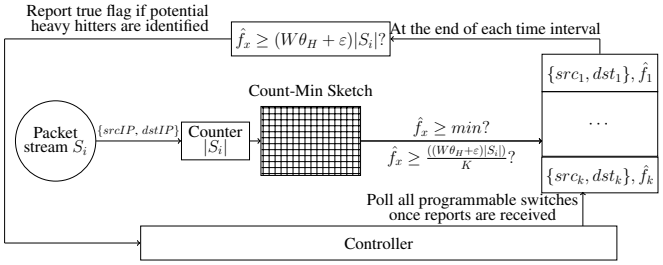


Fig. 4. Scheme of the proposed network-wide heavy-hitter detection strategy

Given:

- A heavy-hitter identification fraction θ_H $(0 < \theta_H < 1)$;
- A time interval T_{int};
- The set of unique packets S transmitted in the network in the time interval T_{int};

Identify the set of flows which are network-wide heavy hitters (HH), i.e., carry in the time interval T_{int} a number of packets larger than the global threshold $\theta_H |S|_{tot}$, where $|S|_{tot}$ is the whole network volume, i.e., the number of transmitted unique packets.

B. Algorithm in programmable switches (Data plane)

As shown in Fig. 4, when a packet comes into a programmable switch i, a packet counter named $|S_i|$ is updated to count all the incoming packets. A Count-Min Sketch data structure, which is used to store the estimated packet counts for all the flows, is updated to include the information from the current packet, and then it is queried to retrieve the estimated packet count $\hat{f}_x$ for the flow $x \sim$ (`srcIP`$_x$, `dstIP`$_x$) such packet belongs to. This information is used to understand whether the packet belongs to a flow that must be inserted in the sample list.

The flow x is inserted in the sample list if $\hat{f}_x \geq \frac{(W\theta_H+\varepsilon)|S_i|}{K}$, where $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$ is the sampling threshold. The parameters W $(W \geq 1)$ and K $(1 < K)$ affect the size of the sample list: the greater W is or the smaller K is , the smaller the sample list size is, thereby consuming less memory in the switch. However, as we will report in the next subsection, this would reduce the accuracy on the estimation of the overall network volume and on the identification of

Algorithm 2: Network-wide heavy-hitter detection algorithm - Programmable switch $i \in \mathcal{P}$ (Data plane)

Input: Flow stream S, Local minimum min, Heavy-hitters identification fraction θ_H, Local ratio W, Sampling rate K, Count-Min Sketch size $N_h \times N_s$, Time interval T_{int}

Output: $flag$ (`true` if potential heavy hitters are identified in time interval, `false` otherwise)

1 $\varepsilon \leftarrow 1/N_s$
2 **Function** *StoreFlowsInSampleList*:
3 $|S_i| \leftarrow 0$
4 $flag \leftarrow$ `false`
5 **while** *currentTime* $< T$ **do**
6 **for** *Each received packet belonging to flow* $x \sim (srcIP_x, dstIP_x)$ *received* **do**
7 $|S_i| \leftarrow |S_i| + 1$
8 **if** $\hat{f}_x \geq min$ *and* $\hat{f}_x \geq \frac{(W\theta_H+\varepsilon)|S_i|}{K}$ **then**
9 $SampleList_i(x) \leftarrow \hat{f}_x$
10 **Function** *PotentialHHsDetection*:
11 **if** *currentTime* = T_{int} **then**
12 **for** *Each flow* x *in* $SampleList_i$ **do**
13 **if** $\hat{f}_x < \frac{(W\theta_H+\varepsilon)|S_i|}{K}$ **then**
14 $SampleList_i.remove(x)$
15 **if** $SampleList(x) \geq (W\theta_H + \varepsilon)|S_i|$ **then**
16 $flag \leftarrow$ `true`
17 **return** $flag$
18 **return** $flag$

heavy hitters. Therefore, K (called *sampling rate*) should be carefully set in each programmable switch to store only flows carrying a significant number of packets. ε is instead the biased value caused by Count-Min Sketch (see Section III): we sum $\varepsilon = 1/N_s$ to $W\theta_H$ in order to compensate such bias. The sample list is thus used to dynamically store the packet counts for the most frequent flows crossing the switch. Since at the beginning of each time interval T_{int} the sampling threshold is low, being $|S_i|$ a small value (that can even be lower than 1), flows with very small packet counts would be stored in the sample list. We thus introduce a parameter min operating in conjunction with the sampling threshold: only if packet count of the considered flow is larger than both min and sampling threshold $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$, the flow is inserted in the sample list. This is described in Lines 2-9 of Algorithm 2.

At the end of the time interval T_{int}, $|S_i|$ counts all the incoming packets in the considered time frame. Thus, as shown in Lines 10-12, the algorithm removes from the sample list all those flows with packet count lower than $\frac{(W\theta_H+\varepsilon)|S_i|}{K}$, where $|S_i|$ is the final stored value. This means that the algorithm keeps in the sample list only those flows which have packet counts larger than the final sampling threshold, while discarding the flows with packet counts greater than the temporary threshold dynamically computed and updated within the time interval. Note that the sample list can store at most $\frac{K}{(W\theta_H+\varepsilon)}$ flows, and thus its memory occupation increases as K increase or W decrease, as already discussed above.

Finally, the algorithm evaluates whether potential heavy hitters cross the switch. They are the flows whose packet counts are greater than the switch local threshold, set as $(W\theta_H + \varepsilon)|S_i|$ (Lines 13-14). A true flag is sent to the controller if at least one potential heavy hitter is detected, otherwise no information is sent. Note that the local threshold is similar to the sampling threshold, and just misses in its definition K, which has been introduced to only set the size of the sample list. The primary role of W is instead to set the proportion (or *ratio*) between the local threshold $(W\theta_H + \varepsilon)|S_i|$ and the global threshold $\theta_H|S|_{tot}$, being S_i a local (and smaller) value than $|S|_{tot}$.

C. Algorithm in centralized controller (Control plane)

At the end of the time interval T_{int}, once the controller receives from the programmable switches the reports including the identified potential heavy hitters, it polls all of them to obtain their sample lists. Note that different sample lists can include the estimated packet count for the same flows: this happens if a flow crosses multiple programmable switches. To avoid an overestimation of $|S|_{tot}$, Algorithm 3 makes sure that *(i)* only the minimum-estimated packet count is kept, i.e., the one less overestimated by Count-Min Sketch, and *(ii)* it is stored in a list (i.e., $GlobalSampleList$) including all distinct flows from sample lists (Lines 2-8). The algorithm then sums up the packet counts for all the identified distinct flows and estimates the whole network volume $|S|_{tot}$ (Lines 9-12). If packet counts of flows belonging to *GlobalSampleList* (which for sure includes the potential heavy hitters) are larger than the global threshold $\theta_H|S|_{tot}$, we consider them as network-wide heavy hitters HH (Lines 13-17). At last, the controller triggers the reset of counters in all programmable switches.

VI. Evaluation results

Based on open source implementations of HyperLogLog [25] and Count-Min Sketch [26], we implemented our incremental deployment algorithm and we simulated both our and Harrison's [20] network-wide heavy-hitter detection strategies in Python. In the following the simulation settings are reported.

A. Simulation settings and evaluation metrics

1) Traces and network topology: We divided 50 seconds 2018-passive CAIDA flow trace [27] into 10 time intervals. The programmable switches send reports to the controller when they detect potential heavy hitters at the end of any of those time intervals: in each time interval are transmitted around 2.3 million packets. As testing topology, we adopted a 100-nodes ISP backbone network [28]. A 32-bit cyclic redundancy check (CRC) [29] function was used to randomly assign each packet (characterized by a specific `(srcIP,dstIP)` pair) to a source/destination node couple in the network, and each packet is routed on the shortest path.

2) Tuning parameters: Unless otherwise specified, we set the simulation parameters as reported in Table I.

Algorithm 3: Network-wide heavy-hitter detection algorithm - Centralized controller (Control plane)

Input: Heavy-hitters identification fraction θ_H, Time interval T_{int}, Sample lists $SampleList_i$ from all programmable switches $i \in \mathcal{P}$

Output: Set of network-wide heavy hitters HH in time interval T_{int}

Function *RetrieveDistinctFlowsPacketCounts*:
  for *Each switch* $i \in \mathcal{P}$ **do**
    for *Each flow* x *in* $SampleList_i$ **do**
      if *flow* x *is in* $GlobalSampleList$ **then**
        if $SampleList_i(x) < GlobalSampleList(x)$ **then**
          $GlobalSampleList(x) \leftarrow SampleList_i(x)$
      else
        $GlobalSampleList(x) \leftarrow SampleList_i(x)$

Function *EstimateVolume*:
  $|S|_{tot} \leftarrow 0$
  for *Each flow* x *in GlobalSampleList* **do**
    $|S|_{tot} \leftarrow |S|_{tot} + GlobalSampleList(x)$

Function *GetNetworkWideHH*:
  for *Each flow* x *in GlobalSampleList* **do**
    if $GlobalSampleList(x) \geq \theta_H |S|_{tot}$ **then**
      $HH.add(x)$
  return *HH*

3) Metrics: We set *recall* R and *precision* Pr as key metrics to evaluate our network-wide heavy-hitter detection strategy. They are defined in the following way:

$$R = \frac{Count_{HeavyHitters}^{detected/true}}{Count_{HeavyHitters}^{detected/true} + Count_{HeavyHitters}^{undetected/true}} \quad (1)$$

$$Pr = \frac{Count_{HeavyHitters}^{detected/true}}{Count_{HeavyHitters}^{detected/true} + Count_{HeavyHitters}^{detected/false}} \quad (2)$$

In our evaluations, we consider *F1 score* ($F1$) as compact metric taking into consideration both precision and recall, and measuring the accuracy of our strategy. It is defined in the following way:

$$F1 = \frac{2 \cdot Pr \cdot R}{Pr + R} \quad (3)$$

Additionally, we consider each {flow key, packet count} pair as *unit* to evaluate both consumed *communication overhead* (when sent) and overall *occupied memory* (when stored in programmable switches). All the reported metrics are the average value obtained in the considered 10 time intervals.

B. Evaluation of the incremental deployment algorithm

We compare our *Incremental deployment algorithm*, where programmable switches are used for the detection of network-wide heavy hitters, with three existing algorithms: *Highest closeness*, *Highest betweenness* and *Random locations*. In the Highest closeness algorithm, the switches are ordered according to decreasing closeness, and in a partial deployment of P programmable switches only the top P switches in the list are replaced [30]. The Highest betweenness algorithm behaves in the same way, but betweenness of nodes [31] is evaluated instead of closeness. Both of the algorithms only depend on the network topology, and their underlying assumption is that nodes with highest centrality should be replaced first. Finally, the Random locations algorithm replaces P randomly-selected nodes: we average results over five randomized instances.

TABLE I
SIMULATION PARAMETERS

HyperLogLog Size m	12
Time interval T_{int}	5s
Sampling rate K	10
Local ratio W	1
Heavy-hitter identification fraction θ_H	0.05%
Local minimum min	1
Count-Min sketch size ($N_h \times N_s$)	10000×40

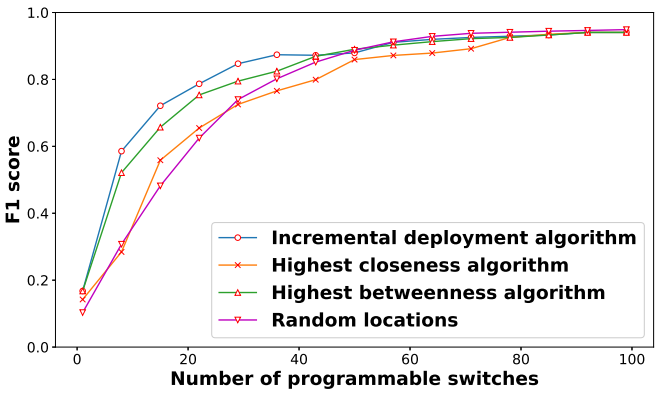
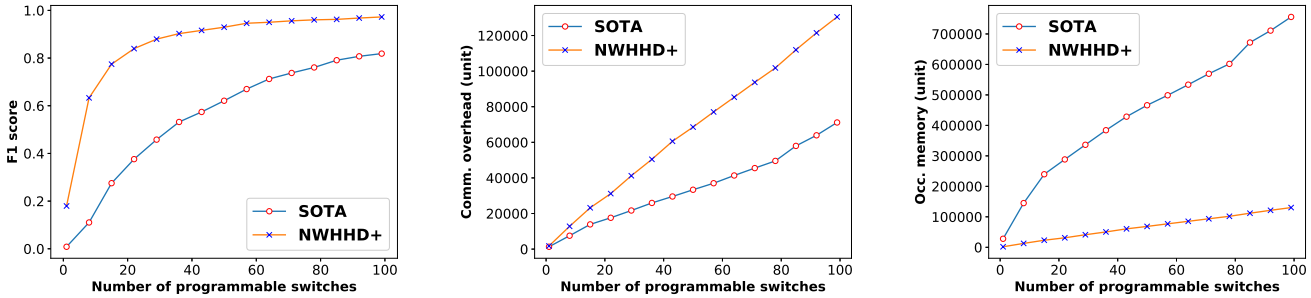


Fig. 5. Performance evaluation of our incremental deployment against some existing algorithms in the detection of network-wide heavy hitters

As shown in Fig. 5, which reports F1 score as a function of the number of deployed programmable switches, our Incremental deployment algorithm allows network operators to deploy a less number of programmable switches while ensuring the same F1 score of the other algorithms. It especially works well when a small number of programmable switches is deployed (i.e., for less than 50 switches), while has comparable performance as the other algorithms when more than half programmable switches are deployed. This means that our strategy of first replacing switches that monitor the highest number of distinct flows effectively improves flow visibility when it is inherently limited.

C. Evaluation of the network-wide heavy-hitter detection strategy in an incremental deployment scenario

We compare the performance of our proposed network-wide heavy-hitter detection strategy, named *NWHHD+* for the sake of brevity, with the state-of-the-art strategy (called *SOTA* in the remainder of the section) proposed by Harrison et al. [20]. In



(a) F1 score comparison (b) Communication overhead comparison (c) Memory occupation comparison

Fig. 6. Performance comparison of NWHHD+ with a state-of-the-art strategy in a partial deployment scenario

order to fairly compare these two strategies, we set the global threshold for SOTA to $T_g = \overline{d}\theta_H|S|_{tot}$, where $\overline{d}$ is the average path length for the flows in all the time intervals, θ_H is fraction for heavy-hitter identification, and $|S|_{tot}$ is the whole network volume. Smoothing parameter is $\alpha = 0.8$ as per [20].

Figure 6(a) shows that NWHHD+, when deploying programmable switches using our incremental deployment algorithm, always leads to higher F1 score than SOTA. This means that NWHHD+ better exploits partial flow information provided by the programmable switches to detect the network-wide heavy hitters.

Figure 6(b) shows instead a comparison on the average-generated communication overhead. It clearly shows that NWHHD+ has a higher communication overhead than SOTA, and the difference becomes even higher as the number of programmable switches increases. This happens because, in NWHHD+, if at least one local heavy hitter is identified in a given time interval (as always happens in our simulations), at the end of it the controller polls all the programmable switches to estimate the global network volume. Conversely, the SOTA strategy coarsely estimates the overall network volume at the controller and polls the programmable switches only if the estimated value is above the global threshold. This coarser estimation allows to save communication overhead but, as shown in Fig. 6(a), has a negative impact on accuracy.

Figure 6(c) shows the average occupied memory in the two strategies. NWHHD+ outperforms SOTA, always occupying much less memory. This happens because NWHHD+ only stores *(i)* the sample list (and not all the {flow key, packet count} pairs, as SOTA does) and *(ii)* one local threshold for all the flows in each programmable switch (while SOTA stores per-flow local thresholds).

Finally, Fig. 7 recalls the simple test described in Section II. What we report in the figure is the F1 score for all the possible 100 deployments for the programmable switch when NWHHD+ is adopted. Compared to Fig. 1, we can see that the F1 score is generally higher in NWHHD+ (as already discussed), and that in most of the cases the peaks in F1 score correspond to IDs of switches that are crossed by a high number of distinct flows (see Fig. 2). This means that NWHHD+ better exploits the distinct flows information than SOTA. This can be even further proven by computing the normalized cross-correlation in $\tau = 0$ [32] between the number of distinct flows and F1 score, which is 13.11 for NWHHD+ and 6.57 for SOTA.

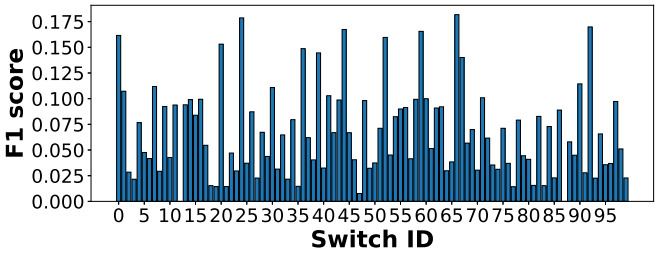


Fig. 7. F1 score of NWHHD+ with single programmable-switch deployment

TABLE II
SENSITIVITY TO W IN THE CASE OF FULL DEPLOYMENT ($K = 10$)

Evaluated metrics	SOTA	NWHHD+			
		W=1	W=3	W=5	W=20
F1 score	0.821	0.948	0.907	0.881	0.823
Communication overhead	71877	131707	60354	41076	13898
Occupied memory	760042	131608	60255	40977	13799

D. Evaluation of the network-wide heavy-hitter detection strategy in a full deployment scenario

We evaluate NWHHD+ strategy against SOTA also in a *full deployment scenario*, i.e., when all the legacy devices have been replaced with programmable switches. In NWHHD+, we have introduced two parameters, i.e., ratio W and sampling rate K, which allow network operators to explore the trade-off between F1 score, communication overhead and memory occupation. Tables II and III show the sensitivity of NWHHD+ to W and K, and a performance comparison with SOTA. Note that, in the two tables, the columns related to $W = 1$ and $K = 10$ report results for the same settings we used in previous subsections on partial deployment, showing (as expected) a much higher F1 score and lower memory consumption than SOTA, but greater communication overhead. The tables also show that by properly tuning W and K it is possible to get a desired performance trade-off among F1 score, communication overhead and memory occupation.

TABLE III
SENSITIVITY TO K IN THE CASE OF FULL DEPLOYMENT ($W = 1$)

Evaluated metrics	SOTA	NWHHD+			
		K=1.2	K=10	K=20	K=100
F1 score	0.821	0.846	0.948	0.970	0.998
Communication overhead	71877	24760	131707	218370	570956
Occupied memory	760042	24661	131608	218264	570875

As shown in Table II, an increase of W leads to a decrease in the sample list size, which means that less {flow key, packet count} pairs are reported to the controller when the switches are polled (i.e., less communication overhead). Additionally, a decrease in the sample list size means that less memory is occupied in the switches. Intuitively, as side effect, the detection accuracy is affected (i.e., lower F1 score). Table II also shows that with $W = 20$, NWHHD+ and SOTA have comparable F1 score, but NWHHD+ leads to a significant reduction of both memory occupation and communication overhead.

Similar results can be obtained by properly tuning K (Table III). An increase of K leads to a smaller sample list and, consequently, to less memory consumption. Moreover, communication overhead is also reduced, because less information is sent when the switches are polled by the controller. Also in this case, with $K = 1.2$, NWHHD+ and SOTA have similar F1 score, but NWHHD+ considerably reduces communication overhead and memory occupation.

Note that such a tuning of W and K leads to analogous trends also in the case of partial deployment, but we omit a quantitative evaluation for the sake of conciseness.

VII. RELATED WORK

A. Partial deployment of SDN solutions in ISP networks

The appearance of SDN simplifies the network management an enhances the flexibility of the network. However, currently it is not feasible to upgrade all legacy switches to SDN switches due to the limitation of budgets and operational burdens, so the current trend for network operators is to deploy a limited number of SDN switches and make the network best work in a hybrid environment. A good strategy for partial SDN deployment is thus needed to cost-effectively bring benefits to ISPs. Unfortunately, obtaining the best partial deployment of SDN switches is a NP-hard problem [33]. In literature, most of the work focuses on the problem of partial deployment of OpenFlow switches [34] in legacy infrastructures, and either Integer Linear Programming (ILP) [35] or incremental deployment heuristic algorithms [33][36][37][38] have been adopted to solve such problem, focusing on interoperability and routing issues in a hybrid environment while achieving the best load balancing or maximizing the throughput. Incremental deployment heuristic strategies are a good approach to solve the problem of partial deployment, since they aim at iteratively replacing legacy equipment by ensuring local optimal performance. However, the previous work neither takes into account the problem of incremental deployment of programmable switches in a legacy infrastructure to improve network monitoring performance nor proposes a solution for topological placement of programmable switches: with our paper, we try to fill this gap.

B. Network-wide heavy-hitter detection in the data plane

In the last years, many strategies have been proposed to monitor heavy hitters directly in the data plane by exploiting the flexibility of programmable switches. Some among them are OpenSketch [39], UnivMon [14], Elastic Skecth [16], FlowRadar [15] and HashPipe [18]. However, all of them only focused on heavy-hitter detection at a single SDN switch, but this is not enough for heavy-hitter detection in large networks, since some heavy hitters may be undetected or wrongly detected by relying on limited information at a single location.

Thus, the concept of *network-wide heavy hitter* has been introduced in literature [19]. A network-wide heavy hitter uses distributed information, which can be made available by programmable switches, to accurately and effectively monitor heavy hitters from a global perspective. Harrison et al. [19] and Basat et al. [20] have proposed two different strategies to monitor network-wide heavy hitters. In Harrison's strategy [20], at the end of each time interval, if any heavy hitter has been detected through a local threshold-based mechanism, the controller polls the programmable switches and uses a different (global) threshold-based mechanism to decide whether local heavy hitters are network-wide heavy hitters or not. However, in their strategy, packets belonging to the same flow are counted multiple times by different switches, and this duplicated information is not discarded by the controller while estimating network-wide heavy hitters: for this reason, it is very difficult to correctly set the global threshold in their strategy. Basat's work [19] provides a solid method for network-wide heavy-hitter detection by using a data streaming model, but the introduced communication overhead and the occupied memory are significant. Another key limitation is that the hash functions their strategy needs do not exist in practice. Our network-wide heavy-hitter detection strategy is similar to the one proposed by Harrison et al., but we define new and more intuitive local and global thresholds and we exploit information on distinct flows to prevent duplicate counting of packets, thus reducing the occupied memory in programmable switches and improving monitoring performance.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented a new greedy algorithm for an effective incremental deployment of SDN programmable switches in legacy infrastructures that aims at monitoring as many distinct network flows as possible. This algorithm best supports monitoring tasks such as heavy-hitter detection when only a limited number of legacy devices can be replaced with programmable switches. We also proposed a novel network-wide heavy-hitter detection strategy which works well in conjunction with our incremental deployment approach. Both the incremental deployment algorithm and the network-wide

heavy-hitter detection strategy were proved to outperform existing approaches. By adopting our incremental deployment algorithm, network operators can ensure very good monitoring performance by replacing less than half of the legacy devices in the network. Moreover, our network-wide heavy-hitter detection strategy outperforms an existing approach both when only a limited number of programmable switches is deployed and when the network is entirely upgraded, since it allows network operators to strike a balance between heavy-hitter detection accuracy, communication overhead and occupied memory.

As future work, we intend to implement our network-wide heavy-hitter detection strategy in a testbed composed of three programmable P4 switches, and thus test it in a real environment. Furthermore, we also plan to extend our strategy to execute a wider range of network-wide monitoring tasks, such as heavy change detection and entropy estimation.

REFERENCES

[1] N. Duffield, C. Lund, and M. Thorup, "Charging from Sampled Network Usage," in *Internet Measurement Workshop (IMW)*. ACM, 2001.

[2] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2002.

[3] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving Traffic Demands for Operational IP Networks: Methodology and Experience," in *Transactions on Networking, Volume: 9, Issue: 3*. IEEE/ACM, 2001.

[4] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *COnference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2011.

[5] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing Network-wide Traffic Anomalies," in *Computer Communication Review, Volume: 34, Issue: 4*. ACM, 2004.

[6] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4. ACM, 2005, pp. 217–228.

[7] Y. Xie, V. Sekar, D. A. Maltz, M. K. Reiter, and H. Zhang, "Worm Origin Identification Using Random Moonwalks," in *Security and Privacy (SP)*. IEEE Computer Society, 2005.

[8] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum, "New Streaming Algorithms for Fast Detection of Superspreaders," in *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2005.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.

[10] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central control over distributed routing," in *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.

[11] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever, "An industrial-scale software defined internet exchange point," in *Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2016.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *Computer Communication Review*, vol. 38, no. 2, 2008.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[14] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 101–114.

[15] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers." in *Nsdi*, 2016, pp. 311–324.

[16] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 561–575.

[17] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 113–126.

[18] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 164–176.

[19] R. B. Basat, G. Einziger, S. L. Feibish, J. Moraney, and D. Raz, "Network-wide routing-oblivious heavy hitters," in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. ACM, 2018, pp. 66–73.

[20] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proceedings of the Symposium on SDN Research*. ACM, 2018, p. 8.

[21] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156.

[22] G. Cormode, "Count-min sketch," in *Encyclopedia of Database Systems*. Springer, 2009, pp. 511–516.

[23] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[24] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central control over distributed routing," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787497

[25] "Python implementation of hyperloglog," https://github.com/ekzhu/datasketch.

[26] "Python implementation of count-min sketch," https://github.com/rafacarrascosa/countminsketch.

[27] "Caida ucsd anonymized internet traces dataset - [passive-2018]," http://www.caida.org/data/passive/passive_dataset.xml.

[28] N. Spring, R. Mahajan, and D. Wetherall, "Measuring isp topologies with rocketfuel," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4, pp. 133–145, 2002.

[29] G. Castagnoli, S. Brauer, and M. Herrmann, "Optimization of cyclic redundancy-check codes with 24 and 32 parity bits," *IEEE Transactions on Communications*, vol. 41, no. 6, pp. 883–892, 1993.

[30] K. Okamoto, W. Chen, and X.-Y. Li, "Ranking of closeness centrality for large-scale social networks," in *International Workshop on Frontiers in Algorithmics*. Springer, 2008, pp. 186–195.

[31] M. E. Newman, "Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality," *Physical review E*, vol. 64, no. 1, p. 016132, 2001.

[32] *Convolution and Correlation*. John Wiley & Sons, Ltd, 2015, ch. 3, pp. 21–36.

[33] D. K. Hong, Y. Ma, S. Banerjee, and Z. M. Mao, "Incremental deployment of sdn in hybrid enterprise and isp networks," in *Proceedings of the Symposium on SDN Research*. ACM, 2016, p. 1.

[34] "Openflow switch," http://www.openflowswitch.org/.

[35] M. Markovitch and S. Schmid, "Shear: A highly available and flexible network architecture marrying distributed and logically centralized control planes," 2015.

[36] H. Xu, X.-Y. Li, L. Huang, H. Deng, H. Huang, and H. Wang, "Incremental deployment and throughput maximization routing for a hybrid sdn," *IEEE/ACM Transactions on Networking (TON)*, vol. 25, no. 3, pp. 1861–1875, 2017.

[37] M. Huang and W. Liang, "Incremental sdn-enabled switch deployment for hybrid software-defined networks," in *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*. IEEE, 2017, pp. 1–6.

[38] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann, "Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks," in *USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 333–345.

[39] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch." in *NSDI*, vol. 13, 2013, pp. 29–42.