Sampling-on-Demand in SDN

Reuven Cohen Evgeny Moroshko
Department of Computer Science
Technion–Israel Institute of Technology
Haifa 32000, Israel

***Abstract*—Sampling is an expensive network resource, because switches and routers are able to sample only a small fraction of the traffic they receive. Modern switches and routers perform uniform packet sampling, which has several major drawbacks: (i) the same flow might be unnecessarily sampled multiple times in different switches; (ii) all the flows traversing a switch whose sampling module is activated are sampled at the same rate; (iii) the sampling rate is fixed, even if the volume of the traffic changes. For the first time, we propose a sampling-on-demand monitoring framework. The proposed framework, presented as a component of SDN (Software Defined Network), adds a Sampling Management Module to the SDN controller. This module allows the controller to determine the sampling rate of each flow at each switch according to the monitoring goals of the network operator, while taking into account the monitoring capabilities of the switch. As part of the proposed framework, the paper defines a new optimization problem called SAP (Sampling Allocation Problem), which has to be solved by the Sampling Management Module in order to maximize the total sampling utility. The paper presents online and offline algorithms for solving this problem. It also presents three real network management applications, executed over Mininet, which are shown to significantly benefit from the proposed framework.**

I. Introduction

Network monitoring plays a significant role in network management. It is used for a variety of applications such as QOS, billing, traffic engineering, security and anomaly detection. While some of these applications require only flow statistics, many require more specific packet-level information. This requirement is fulfilled by having the network switches copy a fraction of the packets of specific flows and forward these packets to a monitoring device for further analysis. This process is known as sampling.

Sampling is an expensive network resource, because each switch is able to sample only a small portion of the traffic it receives. In [32] it is shown that for a given input traffic volume, there is a maximum sampling rate above which the performance of the switch degrades. Although the sampling function is usually performed by the switch's ASIC, it also consumes resources from the switch CPU. Hence, collecting sampled data packets may become a scalability issue, especially in switches with high speed links.

Modern switches include monitoring modules, such as sFlow [5] and NetFlow [3], which use uniform packet sampling In both sampling methods, the sampling rate is determined per switch, usually while taking into account the link speed. This approach has several major drawbacks:

This research was partially funded by the Office of the Chief Scientist of the Israel Ministry of Economy under the Neptune generic research project. Neptune is the Israeli consortium for network programming.

(i) The same flow might be sampled multiple times in different switches not because the management application requires it, but because the flow traverses several switches whose sampling module has been activated, resulting in significant waste of resources.
(ii) All the flows traversing a switch whose sampling module is activated are sampled at the same rate, although the management application might only need to see samples of certain flows, and not necessarily at the same rate.
(iii) The switch samples packets at a predetermined rate, but the volume of the traffic may change. Consequently, when low volume traffic traverses the switch, the switch may not utilize its entire sampling capability, and when high volume traffic traverses the switch, the switch sampling resources may be exhausted. Hence, switch sampling does not adapt to changing traffic volumes, and it is usually activated at a much lower rate than its actual capability.

Previous works evaluate the impact of sampling on specific applications, such as volume anomaly detection and port scans [12], [21], [24], [25], [27]. They show that the effect of the sampling rate on important metrics, such as false positive and false negative rates, is not linear. Hence, when specifying monitoring goals, it could be beneficial to indicate the importance of monitoring one flow over another, and the impact of the monitoring rate on the monitoring goal. It would also be beneficial for the monitoring application to determine the sampling rate of each flow based on this information.

For the first time, we propose a sampling-on-demand monitoring framework. The proposed framework allows the controller to determine the sampling rate of each flow at each switch according to the monitoring goals of the network operator, while taking into account the monitoring capabilities of each switch. The new framework is presented in the context of SDNs (Software Defined Networks), because it is very well suited to such networks. The current OpenFlow Specification (1.5) [7] does not support sampling, but only flow-based monitoring using the *OFPAT_OUTPUT* action. Previous work [30] has already proposed to add a new OpenFlow action that asks specific switches to sample specific flows, but we are the first to propose a model and a mechanism that allow the controller to *determine* which flows should be sampled in every switch and in what rate.

In the proposed framework, every switch has a *sampling capacity* attribute, which indicates the number of packets per second the switch can sample without compromising its performance. For example, some Brocade switches are limited

to 50 samples per second, to avoid CPU bottlenecks [2], and Juniper limits its EX series switches to 300 samples per second per physical interface [4].

The proposed sampling-on-demand framework uses utility functions to specify monitoring goals. These functions, as well as the sampling capacity of all switches, are known to the network (SDN) controller in advance. The controller uses this information to make ongoing centralized decisions about which flows to sample in each switch, and at what rate. The flows to be sampled, the sampling rates, and the locations are determined by the controller such that the total network sampling utility is maximized without exceeding the sampling capacity constraint of each switch. To save sampling resources, the proposed framework guarantees that flows are sampled at most once. An extension of the proposed framework allows sampling the same flow by multiple switches, when this is required by the monitoring application. Sampling before and after a firewall, to see which flows are affected by the firewall's discard rules, is one example of such a case.

As part of the proposed framework, we define a new problem: the Sampling Allocation Problem (SAP), which should be solved by the network controller in order to maximize the total utility of the sampling. We propose online and offline algorithms for solving this problem.

The rest of the paper is organized as follows. In Section II we discuss related work. Section III presents the design of the proposed framework. In Section IV we define and study the sampling allocation problem, and present algorithms for solving it. In Section V we evaluate the proposed framework using Mininet. Section VI evaluates the performance of the proposed algorithms. Finally, we conclude in Section VII.

II. Related Work

In [18], the authors address some of the shortcomings of Sampled NetFlow such as the static sampling rate and the aggregation of flow records, which exhaust the memory during flooding attacks. They propose an improved version of NetFlow, which includes a mechanism for adapting the sampling rate to the traffic mix.

In [32], the authors present OpenSample, a sampling-based monitoring system. OpenSample uses sFlow to obtain packet samples. It reconstructs flow statistics from the samples and estimates port utilization. OpenSample creates a snapshot of the network every 100ms. Each snapshot includes the utilization for every switch port and the list of detected elephant flows. Applications can query the snapshots through an API. Since OpenSample only provides information regarding port utilization and elephant flows, it is mostly suitable for traffic engineering applications and not for applications that require deep packet inspection. In addition, since OpenSample is based on sFlow sampling, the sampling rate is predefined and fixed for each switch. Hence, sampling resources are not well utilized.

In [28], the authors present a framework called FlexSample. FlexSample allows the network operator to specify characteristics of traffic subpopulations, such as packets with a specific source IP address, and define for each subpopulation a budget. FlexSample uses fast counter arrays to determine whether a packet belongs to a subpopulation. If it does, FlexSample samples the packet with a probability proportional to the subpopulations budget. FlexSample allows network operators to define how the sampling rate is divided between different subpopulations, but not to specify the sampling rate. Hence, choosing a sampling rate that fits the sampling capability of the switch remains a challenge. In addition, FlexSample uses m-dimensional arrays for storing the sampling budget, where m is proportional to the number of defined subpopulations, which may exhaust the switch limited RAM.

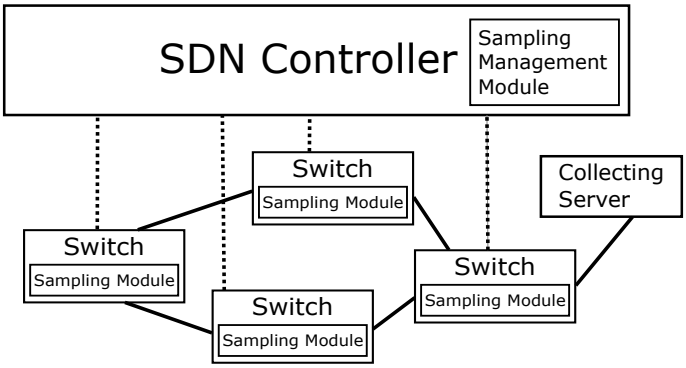


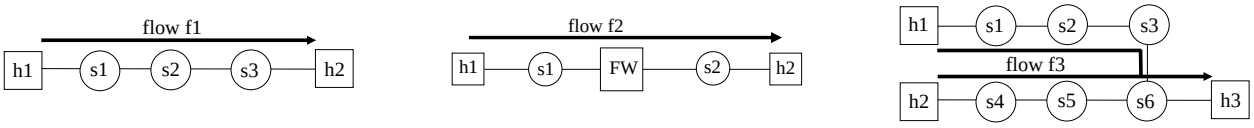
Fig. 1: The components of the proposed framework: the SDN controller uses OpenFlow to control the various switches

In [29], the authors present cSamp — a system-wide framework for flow monitoring. This framework computes a sampling manifest, which states what fraction of each flow is sampled in each switch. cSamp aims to maximize the flow coverage while meeting the switch sampling constraints. It also provides a hash based coordination method that ensures that the same packet is not sampled by multiple switches. In cSamp, all the flows and their rates are assumed to be known in advance. In addition, cSamp allows the network operator to specify the monitoring goal using a single parameter α, which indicates the desired minimum coverage for each flow. Hence, it does not support the specification of different monitoring goals for different flows.

III. The Proposed Framework

We consider an SDN network with three components, as shown in Figure 1. The first is a Sampling Management Module (SMM), which is a controller application. The second is a Sampling Module, which is added to some or all network switches/routers. The third is a "collecting server", one or more of which are located in the network in order to collect and process the sampled packets.

Each sampled packet is encapsulated in a UDP packet and sent to a collecting server. Each switch is configured with the IP address of the collecting server to which it sends its sampled packets. A simple extension allows a different collecting server to be defined per flow. The proposed framework performs monitoring at the granularity of a flow table entry. This means that all the packets corresponding to the same OpenFlow



(a) A [src,dest,*,*,*] flow (b) A [src,dest,*,*,*] flow via a firewall (c) a [*,dest,*,*,*] flow

Fig. 2: Examples of different 5-tuple flow configurations, which are likely to result in different utility functions

table entry are sampled at the same rate. Sampling can be deterministic or probabilistic. In the first case, one of every r packets is sampled, where r is the sampling rate. In the second case, every packet is sampled with probability $\frac{1}{r}$.

We define a new OpenFlow message called *OFPT_RATE_MOD*. This message is sent by the *Sampling Management Module (SMM)* to the switches, and it indicates which sampling rate should be used for each flow. This new message is added to the OpenFlow implementation in the switches and in the controller. To implement sampling-on-demand in an OpenFlow switch, we need to add a new Match/Action table that will include a key and one of the following possible actions: replicate to port, replicate sample to port and encapsulate. The "encapsulate" action should include support for several tunneling mechanisms, including VLAN, VxLan and IP.

Since the SMM is a controller application, it learns the exact network topology using the OpenFlow Discovery Protocol (OFDP). In addition, the controller learns which flows are active by keeping track of the OpenFlow *OFPT_PACKET_IN* and *OFPT_FLOW_REMOVED* messages, which are sent by the switches when a new flow is added or removed.

The proposed framework uses a utility function [15] that reflects the monitoring goal of each flow. As already indicated, a flow can be sampled in one of many switches, and the utility function of sampling a flow in one switch is not necessarily identical to the utility function of sampling the same flow in another switch.

Using OpenFlow, every 5-tuple connection can, in theory, be considered as a different flow. However, this approach does not scale because it requires allocating an OpenFlow switch entry to every connection. Therefore, wildcards are commonly used to represent most of the tuples associated with each flow entry, in which case many TCP connections are mapped to the same flow entry. For example, the Openflow entry associated with [*,dest,*,80,TCP] handles all the TCP connections established with a certain IP address (dest) in the same way[1]. In such a case, an important network management task is estimating the number of TCP connections (or micro-flows) "hidden" behind each wildcarded flow [36]. With this task in mind, we will now discuss the relationship between the monitoring application, the network topology, and the utility function.

In Figure 2(a), a simple flow from h_1 to h_2 traverses three switches: s_1, s_2, and s_3. For such a flow, it would make sense to use the same utility function in all three switches, because there should be no difference between the packets of the flow as observed by each switch.

In Figure 2(b), flow f_2 traverses switches s_1 and s_2, and a firewall (FW). Since the firewall may block or modify some of the packets of f_2, sampling this flow before the firewall in s_1 is different than sampling it after the firewall in s_2. In particular, sampling in s_1 could reveal connections that are dropped by the firewall and are therefore not revealed by sampling in s_2. Hence, the controller should assign higher utility to sampling f_2 in s_1 than in s_2. In fact, if the management application needs to know the number of connections that are dropped by the firewall, it would be beneficial to sample this flow twice: once before and once after the firewall. Sampling of the same flow in multiple switches is discussed in Section IV-C.

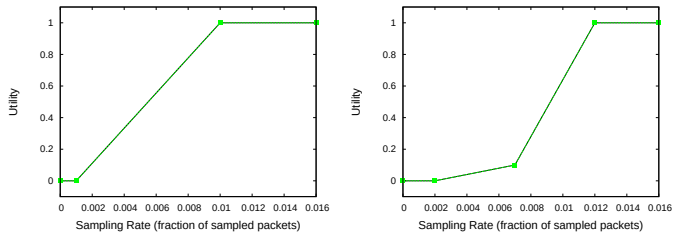
In Figure 2(c), flow f_3 is defined by a destination address only, e.g., all the flows towards a certain web server located in h_3. Since switches $s_1 - s_5$ do not receive all the packets of this flow, the only way to discover all the connections is by sampling this flow in s_6. Hence, the utility of sampling this flow in switches $s_1 - s_5$ should be significantly smaller than the utility of sampling it in s_6.

The utility function is a mapping between a set of possible sampling rates and the interval $[0, 1]$. More formally, a utility function $U_{f(s,r)}$ indicates the "utility to the system" for sampling flow f in switch s using rate r.

Deciding the utility to be assigned to each rate is a challenge. It usually requires evaluating the performance of the application with different sampling rates and assigning a utility in proportion to the evaluated performance. For example, consider again a network management application that estimates the number of micro-flows associated with a certain OpenFlow entry. One can try different sampling rates and, for each rate, run a statistical algorithm that estimates the number of micro-flows from the samples. Then, compare the estimate corresponding to each sampling rate to the real number of micro-flows, and assign a utility in reverse proportion to the estimation error. We use this algorithm in our evaluation section (Section V) with very good results.

Another approach is to understand the application, develop several possible functions and compare their performance. In general, a utility function is always a monotonically increasing function, because high rate sampling is always better than low rate sampling. We found that in most cases the utility function can be represented by a piecewise linear function (Figure 3), because increasing the sampling rate is translated into proportional improvement in the precision of the estimated parameter, but a higher sampling rate affects the precision differently in different ranges: sometimes the improvement is

[1]The 5-tuple represents the source IP address, destination IP address, source port number, destination port number and protocol field.



(a) (b)

Fig. 3: An example of two piecewise linear utility functions, each suitable for a different monitoring goal

significant and sometimes it is marginal.

As an example, suppose that an OpenFlow entry corresponds to all the packets that enter a certain Autonomous System (AS) from a neighboring AS. Let the two ASs be AS1 and AS2 respectively. The operator of AS1 wants to verify that AS2 does not violate the BGP peering relationship. To this end, it needs to verify that the IP destination addresses in the received packets belong to AS1 or its customers. This can be done using a relatively low sampling rate, say 0.001. Increasing the rate to 0.005 would improve the precision of the detection algorithm, but increasing it beyond 0.01 would yield little benefit. This utility function is shown in Figure 3(a).

On the other hand, the utility function in Figure 3(b) is more suitable for a monitoring application that needs to detect port scanning. In this case, the application will prefer high sampling rates, because low rates will result in only some of the flows being sampled. In Section V we study these and other utility functions in greater detail.

IV. The Sampling Allocation Problem (SAP)

A *sampling allocation* is a mapping that indicates which flow should be sampled by which switch and at what rate. In this section we define an optimization problem for finding the optimal sampling allocation, the one that maximizes the total utility while not exceeding the sampling capacity constraints of the switches. We define offline and online versions of the problem and propose efficient algorithms for solving them. We then address an extension for the case where it is beneficial to sample the same flow in multiple switches.

A. Offline SAP

As indicated in Section III, in the proposed framework the SMM determines which flows should be sampled by each switch, and at what sampling rate. This is an online problem, which has to be solved each time a flow enters or leaves the network. We first define and study the offline version of the problem, assuming that all the flows enter the network together. Solving the offline version allows us to gain insight into the problem and obtain a benchmark for the performance of our online algorithm.

Problem 1 (The Offline Sampling Allocation Problem (SAP))

Instance: A set S of network switches. For each switch $s \in S$, c_s is the sampling capacity of s in packets per second (pps). Also given is a set F of flows, with the following information for each flow $f \in F$:

- d_f - the estimated packet rate (packets per second, or pps) sent by f.
- P_f - the path of flow $f \in F$, i.e., the set of switches it traverses.
- R - a set of possible sampling rates supported by the switches.
- a utility function $U_{f(s,r)}$, which indicates the "utility to the system" for sampling flow f in switch s using rate $r \in R$.

Objective: Find a feasible sampling allocation that maximizes the total utility. A feasible sampling allocation is a collection of 3-tuples $[switch, flow, rate]$ that fulfills the following requirements:

(a) for every 3-tuple $[s, f, r]$, s is a switch traversed by the flow f.

(b) the total sampling rate required from each switch $s \in S$ does not exceed its maximum sampling capability, namely, $\sum_{[s,f,r] \in T(s)} d_f \cdot r \leq c_s$, where $T(s)$ is the set of tuples for switch s.

We first assume that the rate of each flow is known to the SMM, and then discuss how to handle rate uncertainties.

When the utility function is a discrete function, SAP can be shown to be equivalent to MC-GAP [14], which is an extension of the Generalized Assignment Problem (GAP). (GAP, in its turn, is an extension of the well-known Knapsack problem). The input for GAP is a set B of bins and a set I of items. Each bin has a size, and each item has a size and a utility. The objective of GAP is to find a subset $U \subseteq S$ of items that have a feasible packing in B, such that the utility of U is maximum.

MC-GAP extends GAP by associating multiple configurations with each item and seeking a collection of configurations, at most one from each item, which can be packed into several bins (knapsacks) without exceeding their capacity. Formally, an instance of MC-GAP is a triplet (B, I, C) and a $3D$ utility matrix P, where B is a set of bins (knapsacks), I is a set of items, C is a set of configurations, and P is a $|I| \times |C| \times |B|$ matrix that indicates the utility and size for each item in each bin using each configuration. The objective is to find a subset $U \subseteq (I \times C)$ of [item, configuration] pairs that has a feasible packing in B, such that each item is chosen at most once, using one of its configurations, and the total utility is maximized.

To transform an instance of SAP to an instance of MC-GAP and vice versa, we represent each SAP switch as an MC-GAP bin whose size is equal to the sampling capacity of the switch. We represent each SAP flow as an MC-GAP item, and each SAP sampling rate as an MC-GAP configuration. The utility of sampling a SAP flow in a switch using a specific sampling rate is represented by the value in the MC-GAP utility matrix for the corresponding [bin, item, configuration].

The Multiple Choice Knapsack Problem (MCKP) is a generalization of the classic Knapsack problem. In MCKP,

there are several classes of items, each having a weight and a utility. The goal is to choose exactly one item from each class, such that the total utility is maximized and the knapsack weight constraint holds. Although MCKP is NP-hard [23], it has efficient approximation algorithms [11] and an optimal pseudo-polynomial time algorithm [23]. In particular, it is shown in [14] that any α-approximation for the Multiple Choice Knapsack Problem (MCKP) [23] can be transformed into a $(1+\alpha)$-approximation for MC-GAP.

We take advantage of the equivalence between SAP and MC-GAP, and use Algorithm ALG_{MC-GAP} from [14] to solve the offline SAP. Algorithm 1 below gives a high-level description of our proposed offline sampling allocation.

Algorithm 1 Offline Sampling Allocation

1. Transform the SAP instance into an MC-GAP instance.
2. Solve the MC-GAP instance using the algorithm proposed in [14].
3. Transform the solution for MC-GAP into a solution for SAP.

The algorithm for solving MC-GAP in [14] is a $(1+\alpha)$-approximation algorithm, which extends the one presented in [16] for solving GAP. The algorithm can be implemented iteratively, with running time of $O(|B|\cdot T_{\text{MCKP}}(|C|,|I|)+|B|\cdot |I|\cdot |C|)$, where $T_{\text{MCKP}}(|C|,|I|)$ is the running time of the MCKP algorithm used in Step 2. Hence, the running time of Algorithm 1 is $O(|S|\cdot T_{\text{MCKP}}(|F|,|R|)+|S|\cdot|F|\cdot|R|)$. MCKP can be solved using an efficient greedy algorithm [23] with running time of $O(|F|\cdot|R|\cdot log(|R|)+|F|\cdot|R|\cdot log(|F|\cdot|R|))$.

B. Online SAP

In the online version of SAP, flows are admitted one at a time into the network. When a new flow is admitted, the SMM needs to decide whether to sample it, in which switch, and at what sampling rate. Such a decision has to be made while taking into account all previously admitted flows. In particular, the SMM may need to change the sampling location and/or rate of previous flows in order to efficiently sample the new one. Such a reconfiguration imposes extra overhead both on the SMM and on the switches. To minimize this overhead, we add a constraint where the SMM is allowed to change the sampling rate of existing flows, even to 0 if needed, only in the switch that the SMM assigns to sample the newly admitted flow.

As an example, Figure 4 describes a simple topology with 4 switches and 3 flows. Suppose that flows f_1 and f_2 have already been admitted and are sampled in s_2 and s_4 respectively. When f_3 is admitted, the SMM needs to decide whether to sample it in s_2 or in s_4. If it decides to sample f_3 in s_2, it can reduce the sampling rate of f_1, but it cannot change the sampling location of either f_1 or f_2, although changing the sampling location of f_1 to s_1 or s_4 is likely to increase the total utility.

The above constraint guarantees that the number of sampling control messages exchanged between the SMM and the switches is minimized. This is because when a new flow is admitted, the SMM needs to send at most one control message. This message is sent to the switch chosen for sampling the new flow, and it may also ask this switch to reduce the sampling rate of one or more previously admitted flows.

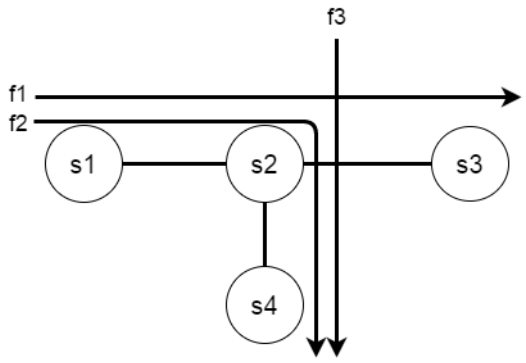


Fig. 4: A topology with 3 flows: flow f_3 is added to the network and only the sampling assignments of s_2 or s_4 can be modified

A formal definition of the Online Sampling Allocation Problem is as follows.

Problem 2 (The Online Sampling Allocation Problem)

Instance: A network as in Problem 1, and a set of flows that are already admitted and sampled. The existing sampling allocation C is feasible. In addition, we are given a new flow f' and the path of this flow.

Objective: Find a feasible sampling allocation C' for the flows in $F\cup\{f'\}$, which maximizes the total utility, such that: (i) except for the new flow f', each sampled flow in C' is sampled in the same switch it was sampled in C; (ii) all the switches use the same sampling rates as in C, except for the switch that samples the new flow f'.

Since online SAP allows changing only the sampling rate of the flows in the switch that samples the new flow, an algorithm for this problem needs to consider only the switches along the path of the new flow. For each switch along this path, such an algorithm determines the increase in total utility if this switch is chosen to sample the new flow. Then, the algorithm chooses the switch for which the utility increase is maximized.

Algorithm 2 presents a formal description of this idea. For each switch traversed by the new flow, the algorithm computes its optimal sampling allocation. To this end, the algorithm solves an MCKP instance for the switch, while taking into account all the flows that this switch already samples, the flows that traverse this switch and are not sampled elsewhere, and the new flow.

Algorithm 2 Online Sampling Allocation

1. For each switch $s\in P_{f'}$, use a solution for MCKP to compute the sampling allocation and the extra utility obtained by sampling f' in s.
2. Choose the switch for which the extra utility is maximum, and return its sampling allocation.

Step 1 of Algorithm 2 can be performed using a greedy MCKP algorithm [23]. Since Algorithm 2 solves MCKP for each switch along the path of the new flow, its running time is $O(|P_{f'}| \cdot T_{\text{MCKP}}(|F|,|R|))$, where $T_{\text{MCKP}}(|F|,|R|)$ is the running time of the MCKP algorithm.

The sampling allocation for a certain switch also must be calculated when the controller learns about the expiration of a sampled flow. In such a case, the controller re-computes the sampling allocation for the sampling switch using an MCKP algorithm. Thus free sampling resources can be assigned to other flows traversing the same switch.

Recall that the sampling load imposed on a switch by a certain flow is the product of the flow rate and the sampling rate. To compute the sampling allocation vector for each switch, the SMM needs accurate information regarding the rates of the flows. Such information can be obtained after flows are admitted into the network using OpenFlow counters. For example, using openTM[34].

However, the rate of a flow is likely to change over time. To cope with such changes, we propose to reduce the sampling capacity of a switch by a factor of F, where $F < 1$. This would enable the switches to cope with small bandwidth increases without making too many changes. To cope with rapid changes, such as those taking place during a flooding attack, we propose to use a watchdog timer in each switch, which resets every second. Each reset, the number of sampled packets is compared to the switch's sampling capacity, hence guaranteeing that the switch will not exceed its maximum desired capacity.

C. Extended SAP

SAP finds a sampling allocation in which every flow is sampled at most once. However, sampling a flow in multiple switches is sometimes beneficial. Consider, for example, the case where flows pass through a firewall. It would be helpful to sample each such flow twice: once before the firewall and once after. The sampling data can help verify that the firewall rules are correct.

We define an extended version of SAP, called E-SAP, which enables sampling the same flow at most twice. A similar extension can be defined for sampling a flow in more than 2 switches, but the size of the input matrix grows exponentially, and building this matrix becomes intractable. In addition, limiting the number of switches to 2 enables us to develop an online algorithm for E-SAP that provides an optimal allocation in practical time.

In E-SAP, the utility function for each flow is defined for every pair of switches rather than for every single switch. The utility function of flow f for switches $[s_i, s_j]$ and rate r defines the utility to the system of sampling f in switches s_i and s_j using rate r. Theoretically, the model can allow two different sampling rates: one for s_i and one for s_j. However, this extension requires adding another dimension to the utility functions, which significantly increases the size of the framework input and renders the model impractical. Hence, we do not discuss it further.

A formal definition for the offline version of E-SAP is as follows.

Problem 3 (The Offline Extended Sampling Allocation Problem (E-SAP))

Instance: A network as in Problem 1, and a utility function U_f. In this function, $U_{f(s_i,s_j,r)}$ indicates the utility for sampling flow f in switches s_i and s_j using rate r. If $i = j$, the function indicates the utility for sampling f in a single switch i.

Objective: Find a feasible sampling allocation, which maximizes the total utility. A feasible sampling allocation is a collection of 3-tuples $[[switch1, switch2], flow, rate]$ that fulfills the following requirements:

(a) for every 3-tuple $[[s_i, s_j], f, r]$, s_i and s_j are switches traversed by the flow f.
(b) the total sampling rate required from each switch $s \in S$ does not exceed its maximum sampling capability.

The d-Dimensional Multiple-Choice Knapsack Problem (d-MCKP) [23] is a combination of MCKP and the d-Dimensional Knapsack Problem (d-KP) [23]. It is similar to MCKP, except that there are d knapsacks, and each item has a d-dimensional weight vector. The objective is to pack exactly one item from each class such that the total utility is maximized and the constraints on the size of all knapsacks hold. Since it is a generalization of d-KP, which is one of the hardest NP-hard problems, d-MCKP is not only NP-hard but also very hard to approximate. It is also known to be NP-hard in the strong sense [26].

To solve offline E-SAP, we note that it is a special case of d-MCKP where the number of knapsacks is equal to the number of switches in the network, and each item's weight vector has at most 2 non-zero values. Algorithm 3 below is a high-level description of offline E-SAP, which uses a procedure for solving d-MCKP. Many heuristics have been presented for d-MCKP[10], [20], [26], but none provides a performance guarantee. In [35], a dynamic programming algorithm for solving d-KP is presented. Using similar ideas, a dynamic programming algorithm for d-MCKP is presented in [13]. The time complexity of this algorithm renders it impractical when the number of knapsacks is not small. Hence, it is suitable for solving offline E-SAP only when the number of switches in the network is small.

Algorithm 3 Offline Extended Sampling Allocation

1. Transform the E-SAP instance into a d-MCKP instance, where d is the number of switches in the network.
2. Solve the d-MCKP instance, e.g., using dynamic programming [13].
3. Transform the solution for d-MCKP into a solution for E-SAP.

Since Algorithm 3 solves offline E-SAP using a solution for d-MCKP, its running time is equal to the running time of the d-MCKP algorithm used in Step 2. Let $T_{\text{d-MCKP}}(d, C, I)$ be the running time of the d-MCKP algorithm with d knapsacks, C classes and I items. There are $\binom{|S|}{2} + |S|) \cdot |R|$ options

for sampling each flow, where $|S|$ is the number of switches and $|R|$ is the number of possible sampling rates. Hence, the running time of Algorithm 3 is $T_{\text{d-MCKP}}(|S|, |F|, (\binom{|S|}{2} + |S|) \cdot |R|)$, where $|F|$ is the number of flows in the network.

The online version of E-SAP is defined in the same way online SAP was defined earlier. It is solved by Algorithm 4, which extends Algorithm 2. In step 1, the algorithm considers each pair of switches $[s_i, s_j]$ along the path of a new flow, and determines the increase in total utility if the flow is chosen to be sampled in these switches. For this task, the algorithm calculates the optimal sampling allocation for each pair of switches, which is equivalent to solving d-MCKP when the number of knapsacks is 2. Sometimes it would be better to sample the new flow only in one rather than two switches. Such a solution cannot be found by step 1, because in this step the sampling rate of the two switches must be equal. Thus, in step 2 the algorithm also computes the utility gain when the new flow is sampled only once, for each switch along the path. The algorithm then chooses the best solution from steps 1 and 2.

Algorithm 4 Online Extended Sampling Allocation

1. For each pair of switches s_i and s_j along the path of the new flow, use a solution for 2-MCKP to compute the optimal sampling allocation and the extra utility obtained by sampling the new flow in these switches.
2. For each switch s_i along the path of the new flow, use a solution for MCKP to compute the optimal sampling allocation and the extra utility obtained by sampling the new flow only in this switch.
3. Let S be the set of switches (a single switch or a pair of switches) for which the extra utility found by the previous steps is maximum.
4. Update the sampling rates in each switch $s \in S$ according to the new sampling allocation.
5. For each flow f, if f is sampled in a switch $s \in S$ and in a switch $s' \notin S$, update the sampling rate of f in s' according to the new sampling allocation.

Since the algorithm requires that d-MCKP be solved for a small number of knapsacks (d=2), the dynamic programming algorithm presented in [13] can be used.

We now analyze the running time of Algorithm 4. Denote by $T_{\text{MCKP}}(C, I)$ the running time of the MCKP algorithm with C classes and I items. Let n be the number of switches on the path of the new flow. Since there are $\binom{n}{2}$ options for sampling the flow in 2 switches, and n options for sampling the flow in a single switch, the total running time of the algorithm is $\binom{n}{2} \cdot T_{\text{d-MCKP}}(2, |F|, |R|) + n \cdot T_{\text{MCKP}}(|F|, |R|)$.

V. Framework Evaluation using Mininet

A. Evaluation Testbed

We implemented a prototype of our framework using a Mininet network emulator [6] running Openflow 1.3 software switches[2]. We added our new *OFPT_RATE_MOD* message to OpenFlow. Recall that this message allows the controller to configure the sampling rate of each flow.

[2]Some of the code used in this section can be found in https://goo.gl/HPsMjU.

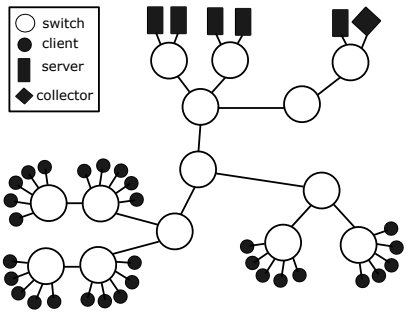


Fig. 5: The Mininet topology for evaluating the proposed framework

Two sampling methods are implemented by the switch: uniform packet sampling, and the proposed flow-based sampling framework. Each switch is also configured with an address of a collector to which the sampled packets are forwarded using UDP encapsulation. Each encapsulated packet is pushed into the switch's pipeline for processing and is then routed towards its collector.

The sampling allocation module is implemented as a standalone Python script. The script configures the switches via the switch management utility (dpctl), which is also modified to support the new OpenFlow *OFPT_RATE_MOD* message. Scapy is used for generating TCP traffic at specific rates between the network switches.

Our experiments are conducted on the topology depicted in Figure 5, with 14 switches and 36 hosts. One of the hosts is configured as a collector, 5 as servers, and 30 as clients.

In all our experiments, when our framework is used, the switch maximum sampling rate is configured to 50 pps (packet per second), because this is the maximum rate the switches in our setup could sustain. When uniform sampling is used, the sampling ratio is set to 1/60, which is the maximum ratio that guarantees that no switch will have to sample more than 50 pps.

B. Estimating the Number of Connections in a Flow

In this section we evaluate the proposed sampling-on-demand framework and algorithms. Our first application example is estimating the number of connections carried by each flow. This is an important management function, which allows the detection of DDoS attacks, deciding whether more resources are needed in the network, and many other network tasks.

We define a connection as a 4-tuple: *[src_ip, dst_ip, src_port, dst_port]*. Hence, the problem of estimating the number of connections is translated into that of estimating the number of distinct 4-tuples (micro-flows) "hidden" behind each (OpenFlow) macro-flow. This problem is known as the cardinality estimation problem (or as the count-distinct problem) [33]. Since the collector receives only samples of each flow, most cardinality estimation algorithms, such as [19], cannot be used because they assume that the entire stream is

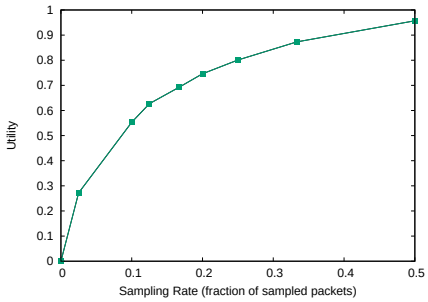


Fig. 6: The utility function for estimating the number of connections in each flow from the flow sample

given to the algorithm as input. For example, if the full stream contains packets from connections c_1, c_2, c_3 and c_4, and the sampled stream contains only packets from connections c_1, c_3 and c_4, a naive algorithm would incorrectly estimate that the number of connections in the full stream is 3 rather than 4. A more sophisticated algorithm, such as the one proposed in [17], is able to estimate the probability of unseen connections and take them into account.

The algorithm in [17] combines sampling with any full-stream cardinality estimation algorithm. We use this method with the HyperLogLog cardinality estimation algorithm [19] to estimate the number of connections in each flow, given only samples from this flow.

The goal of our first experiment is to find a suitable utility function. We use the packet trace from [8] and sample it using multiple sampling rates. We count the number of connections in the unsampled trace. We then apply the estimation algorithm from [17] on each sampled trace. The output of the algorithm is used as an estimation for the number of connections in the unsampled trace. This output is then compared to the real number of connections in the unsampled trace, and the estimation error is calculated. We define the utility for each sampling rate as 1 minus this error. Figure 6 shows the resulting utility function.

We perform two experiments: one with uniform packet sampling, and another with our framework, using the utility function depicted in Figure 6. We define 30 [source, destination] flows in the network. The source of each flow is one of the clients, and the destination is a randomly chosen server. Routing is performed over the shortest paths. Each flow carries between 100 and 3,900 TCP connections between the considered client-server pair. All the connections are initiated roughly at the same time and last 2 minutes. Data packets are sent at a rate of 100pps. We run the algorithm from [17] for each packet trace obtained from the collector, and calculate the percent error for each flow as

$$Error = 100 \cdot \frac{|Real - Estimation|}{Real}.$$

Each experiment is repeated 90 times. Figure 7 shows the average estimation error as a function of the number of connections in each flow, with and without our framework. We can see that our framework yields a slightly smaller estimation error. In most cases the error in our framework is around 3.5%, while without our framework it is typically around 10.5%.

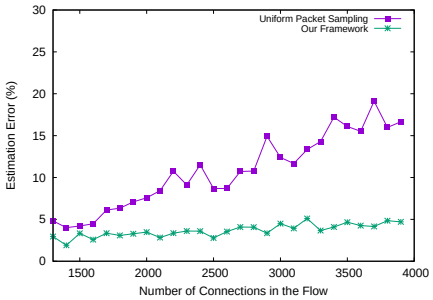


Fig. 7: Estimating the number of connections in a flow: our framework vs. uniform packet sampling

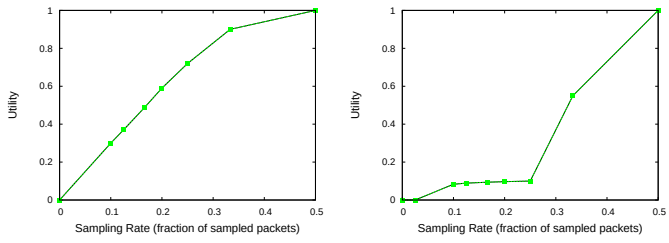
C. Detection of a Port Scanning Attack

Our second application example detects a port scanning attack. In this attack, the attacker probes the victim server for open ports. Such an attack is often used to reveal which services are running on the server. The attacker then uses this information to exploit vulnerable services.

In [22], the authors present a port scanning detection algorithm, called Threshold Random Walk (TRW). In TRW, each host in a packet trace is classified as either a port scanner or benign. TRW requires flow traces that include bidirectional packets, and it is shown to require an extremely low number of observed events to make a decision [31].

For our evaluation, we use a port scanning detection algorithm called TRWSYN [31]. TRWSYN is a variation of TRW, which can work on unidirectional flow traces. We configure each client in the network to establish N connections to one of the servers and send legitimate data. All the connections from a client to a server are considered a single flow, and each flow is routed over its shortest path. From all the clients, 10 are configured to perform port scanning. These 10 hosts send SYN packets to a range of server ports. Each client sends packets at a rate of 100pps.

The experiment is executed 3 times. The first is with uniform packet sampling, the second is with our framework using the utility function depicted in Figure 8(a), and the third is with our framework using the utility function depicted in



(a) (b)

Fig. 8: The utility functions for the second and third applications

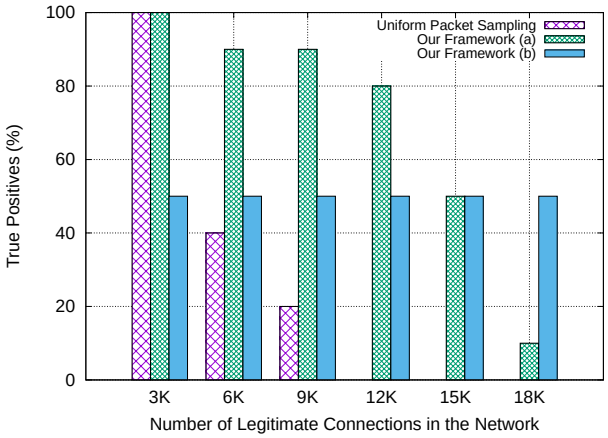


Fig. 9: Comparison of port scanning detection

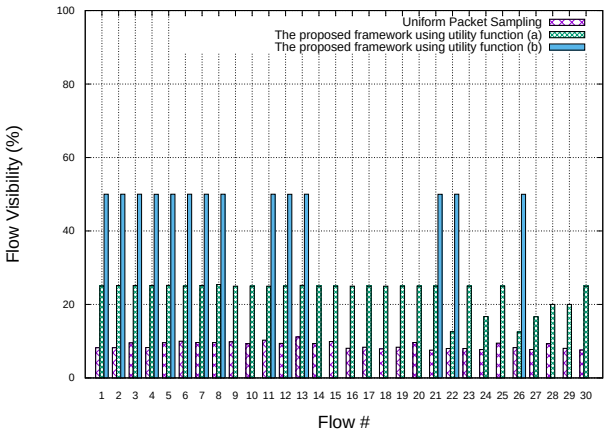


Fig. 10: Flow visibility of different sampling methods

Figure 8(b). The TRWSYN algorithm is applied on the traces sent to the collector during each experiment. The output of this algorithm is a list of clients identified as port scanners. We repeat these 3 experiments while changing the number N of legitimate connections each client established. This increases the noise in the network and makes it harder to detect the port scanners. We then compute for each experiment the number of true positives, namely, clients correctly identified as port scanners, as well as the number of false positives, namely, innocent clients incorrectly identified as port scanners.

Figure 9 shows the true positive rate for different values of N. First, we compare uniform packet sampling to our framework with the utility function of Figure 8(a). The graph shows that when the number of connections in the network is relatively small (3K), both methods perform equally well, and both detect all port scanners. When the number of legitimate connections in the network increases to 6K and to 9K, the ability of uniform sampling to detect the attack dramatically decreases to 40% and 20% respectively. In contrast, our framework is still able to detect 90% of the attacks!

We then compare our proposed framework using the utility functions of Figure 8(a) and Figure 8(b). The former is better when the number of connections is less than 15K; otherwise the latter is better. This is because the percentage of packets associated with port scanning activity decreases when the number of connections is high. Hence, a high sampling rate is required to detect the port scanners. The utility function of Figure 8(b) samples flows at a high rate, at the cost of not sampling other flows at all. Hence, it is able to detect port scanners even when the number of connections is high.

We found no false positives in any of our experiments. This is because TRW produces false positives only when it observes multiple consecutive SYN packets from a benign host, a scenario whose probability is low.

D. Maximizing Flow Visibility

Our third application example measures the flow visibility achieved by the proposed framework. The *visibility* of a flow is defined as the percentage of the flow's packets received by the monitoring application. Many monitoring applications, such as DDoS detection, heavy hitter identification, and port scanner detection, benefit from maximum network visibility [9], [37].

The performance of the proposed framework with respect to flow visibility is compared to the performance of uniform packet sampling, which is the approach used today by sFlow and NetFlow. In uniform packet sampling, each switch samples all received packets using the same sampling rate.

All the packets transmitted from a client to a server are considered as a single flow. Each flow is routed over its shortest path. The collector saves a trace of all the packets it receives. These traces are then used for analyzing the results. The transmission rate of every flow is configured to 100pps. We again perform 3 experiments: (i) using uniform packet sampling; (ii) using our framework, with the utility function depicted in Figure 8(a); (iii) using our framework, with the utility function depicted in Figure 8(b).

Using the traces from the experiments, we calculated the flow visibility for each of the 30 flows, that is, the percentage of each flow's packets received by the collector. Figure 10 shows the visibility of each flow in each experiment. Using our framework with the utility function of Figure 8(a), the average flow visibility is 2.6 times greater than the average flow visibility with uniform packet sampling. Using our framework with the utility function from Figure 8(b), the visibility of flow f_1 is 50% and the visibility of flow f_9 is 0% as opposed to 8.2% for flow f_1 and 9.8% for flow f_9 using uniform packet sampling. So while the visibility of flow f_1 is 6 times greater than with uniform packet sampling, the flow visibility of many other flows is 0. Thus, we conclude that such a utility function is suitable for applications that benefit from high sampling rates, but do not benefit at all from low sampling rates.

E. Discussion

We showed in the previous subsections that the proposed scheme outperforms uniform sampling. There are three main reasons for this result:

(a) Sampling on-demand does not sample the same flow more than once, unless sampling in two locations is profitable

(which was not the case in any of our examples). This allows sampling-on-demand to sample more flows compared to uniform sampling. This helped mainly in the second and the third applications.

(b) The utility function ensures that flows are not sampled using a rate that is higher or lower than necessary. In particular, for some applications, not sampling at all is preferable to sampling at low rate because the sampling budget can then be spent only on management applications that can benefit from it. This helped mainly in the first application.

(c) When the traffic crossing a switch is lower than the maximum switch capacity, sampling-on-demand can sample more than uniform sampling. This mainly helps those applications that benefit from a high sampling rate, namely, port scanning and maximizing flow visibility.

VI. The Performance of the Algorithms

This section evaluates the performance of our online SAP algorithm (Algorithm 2), by comparing its performance to the performance of offline SAP (Algorithm 1), which is taken as a benchmark. Since Algorithm 1 requires solving MC-GAP, which is NP-hard, we use the 2-approximation to MC-GAP proposed in [14].

The evaluation is conducted by simulating a network with 100 switches, generated using Brite [1]. The sampling capacity of each switch is defined as 100pps, and 100 [source, destination] flows are generated. The source and destinations of each flow are randomly chosen, and the flow is established on their shortest paths. We generated 15 variations of the utility function of Figure 8(a), 8(b) and 6. Each flow uses one of these utility functions.

We define the load on the network switches as the number of packets sampled each second in the entire network. The maximal load is therefore defined as the total sampling capacity of all the switches, that is, the maximal number of samples that can be generated in the entire network every second. To simulate a specific network load, we choose the rates of the flows such that their total sum, multiplied by the highest sampling rate, is equal to the required network load.

For each network load, we find a sampling allocation for the 100 flows using Algorithm 1 and Algorithm 2 and calculate the total utility for each algorithm. Since Algorithm 2 is an online algorithm, we assume that it receives the flows one at a time, and we run it for each new flow. The arrival order of the flows is random. This comparison is repeated 2,000 times. Each time 10 flows are replaced by new flows without changing the total load. Then, the average utility of each algorithm is calculated.

Figure 11 shows the average utility increase of Algorithm 2 vs. Algorithm 1, as a function of the network load. It is computed as

$$100 \cdot \frac{OnlineAverage - OfflineAverage}{OfflineAverage}.$$

It is evident that the two algorithms provide very similar average utility when the load is low. Suprisingly, as the load increases, Algorithm 2 slightly outperforms 1, despite its being online whereas Algorithm 1 is offline. The reason is that Algorithm 2 is invoked for every flow, and its overall computational cost is higher than that of Algorithm 1.

We repeated the same analysis when the number of flows arriving to and leaving the network at each iteration was 1 and 50, and received similar results.

While Algorithm 2 yields slightly better results than Algorithm 1, it should be used only for the online problem, when only one connection is admitted at a time. Solving the offline problem for F flows using Algorithm 2 is impractical, since this would require executing this algorithm once per flow.

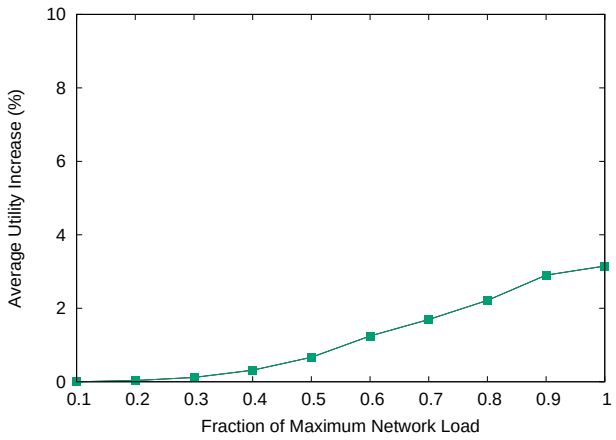


Fig. 11: The performance advantage of Alg. 2 over Alg. 1

We also evaluated the time required for the SMM to calculate the sampling allocation when a new flow arrives. Recall that the sampling allocation is calculated using Algorithm 2, which solves MCKP for each switch in the new flow's path. Figure 12 shows the average running time for solving MCKP for each switch on an Ubuntu Virtual Machine with 4 cores and 2GB of memory, when the utility function from Figure 8(a) is used, and when MCKP greedy [23] is implemented. The total running time required for the SMM to calculate a sampling allocation is the sum of the time required to calculate MCKP for each switch in the new flow's path. For example, if each switch has a sampling capacity of 100pps, each switch already hosts 1K sampled flows, and the new flow traverses 10 switches, the sampling allocation calculation will complete in $10 \cdot 2.55 = 25.5$ milliseconds.

VII. Conclusions

This paper presented a sampling-on-demand monitoring framework. The proposed framework allows the sampling rate of each flow at each switch to be determined according to the monitoring goal of the network operator, while taking into account the monitoring capabilities of each switch. As part of the proposed framework, we defined a new optimization problem called SAP (Sampling Allocation Problem), which has to be solved by the network controller in order to maximize the total utility of the sampling. The paper presented both online and offline algorithms for SAP and evaluated their performance. We evaluated the proposed framework by presenting three real

Number of Flows	Sampling Capacity (pps)	Running Time (mSec)
10	50	0.038
	100	0.035
500	50	1.458
	100	1.617
1K	50	2.812
	100	2.555
10K	50	29.126
	100	28.932

Fig. 12: Running time for solving MCKP for a single switch

network management applications: counting the number of connections carried by each flow, detecting port scanners, and maximizing flow visibility. In these applications we showed that the proposed framework yields excellent performance, significantly outperforming the naive sampling approach used today.

REFERENCES

[1] Brite topology generator. https://www.cs.bu.edu/brite/.

[2] Broacade sFlow configuration considerations. http://www.brocade.com/content/html/en/configuration-guide/fastiron-08040-monitoringguide/GUID-F46B8130-C143-4BDD-AE04-E64821A1A288.html.

[3] Cisco NetFlow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[4] Configuring sFlow technology to monitor network traffic on EX series switches. https://www.juniper.net/documentation/en_US/junos/topics/example/sflow-configuring-ex-series.html.

[5] InMon sFlow. http://www.sflow.org/.

[6] Mininet network emulator. http://mininet.org.

[7] OpenFlow Switch Specification v1.5. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.

[8] UCLA computer science department packet trace. https://lasr.cs.ucla.edu/ddos/traces/public/trace1/tcp/file1.

[9] Y. Afek, A. Bremler-Barr, S. Landau Feibish, and L. Schiff. Sampling and large flow detection in SDN. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 345–346. ACM, 2015.

[10] M. M. Akbar, M. S. Rahman, M. Kaykobad, E. G. Manning, and G. C. Shoja. Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. *Computers & Operations Research*, 33(5):1259–1273, 2006.

[11] M. Bansal and V. Venkaiah. Improved fully polynomial time approximation scheme for the 0-1 multiple-choice knapsack problem. *International Institute of Information Technology Tech Report*, 2004.

[12] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 159–164. ACM, 2006.

[13] R. Cohen and G. Grebla. Multi-dimensional OFDMA scheduling in a wireless network with relay nodes. In *Infocom'2014, Toronto, Canada*, Apr. 2014.

[14] R. Cohen and G. Grebla. Joint scheduling and fast cell selection in OFDMA wireless networks. *IEEE/ACM Transactions on Networking*, 23(1):114–125, 2015.

[15] R. Cohen and L. Katzir. A generic quantitative approach to the scheduling of synchronous packets in a shared uplink wireless channel. *IEEE/ACM Transactions on Networking (TON)*, 15(4):932–943, 2007.

[16] R. Cohen, L. Katzir, and D. Raz. An efficient approximation for the generalized assignment problem. *Information Processing Letters*, 100(4):162–166, 2006.

[17] R. Cohen, L. Katzir, and A. Yehezkel. Cardinality estimation meets good-turing. *arXiv preprint arXiv:1508.06216*, 2015.

[18] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 245–256. ACM, 2004.

[19] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms 2007 (AofA07)*, pages 127–146, 2007.

[20] M. Hifi, M. Michrafy, and A. Sbihi. Algorithms for the multiple-choice multidimensional knapsack problem. *Les Cahiers de la MSE: série bleue*, 31, 2003.

[21] K. Ishibashi, R. Kawahara, M. Tatsuya, T. Kondoh, and S. Asano. Effect of sampling rate and monitoring granularity on anomaly detectability. In *2007 IEEE Global Internet Symposium*, pages 25–30. IEEE, 2007.

[22] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 211–225. IEEE, 2004.

[23] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, 2004.

[24] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 165–176. ACM, 2006.

[25] J. Mai, A. Sridharan, C.-N. Chuah, H. Zang, and T. Ye. Impact of packet sampling on portscan detection. *IEEE Journal on Selected Areas in Communications*, 24(12):2285–2298, 2006.

[26] M. Moser, D. P. Jokanovic, and N. Shiratori. An algorithm for the multidimensional multiple-choice knapsack problem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 80(3):582–589, 1997.

[27] I. Paredes-Oliva, P. Barlet-Ros, and J. Solé-Pareta. Portscan detection with sampled netflow. In *International Workshop on Traffic Monitoring and Analysis*, pages 26–33. Springer, 2009.

[28] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani. Fast monitoring of traffic subpopulations. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, pages 257–270. ACM, 2008.

[29] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. cSamp: A system for network-wide flow monitoring. In *NSDI*, volume 8, pages 233–246, 2008.

[30] S. Shirali-Shahreza and Y. Ganjali. FleXam: flexible sampling extension for monitoring and security applications in OpenFlow. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pages 167–168. ACM, 2013.

[31] A. Sridharan, T. Ye, and S. Bhattacharyya. Connectionless port scan detection on the backbone. In *Proceedings of the Malware Workshop (held in conjunction with IPCCC)*, 2006.

[32] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter. OpenSample: A low-latency, sampling-based measurement platform for commodity SDN. In *IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*, pages 228–237. IEEE, 2014.

[33] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *Proceedings of the 20th International Conference on Data Engineering*, pages 214–225. IEEE, 2004.

[34] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: traffic matrix estimator for openflow networks. In *International Conference on Passive and Active Network Measurement*, pages 201–210. Springer, 2010.

[35] H. M. Weingartner and D. N. Ness. Methods for the solution of the multidimensional 0/1 knapsack problem. *Operations Research*, 15(1):83–103, 1967.

[36] P. Wette and H. Karl. Which flows are hiding behind my wildcard rule?: Adding packet sampling to openflow. *SIGCOMM Comput. Commun. Rev.*

[37] A. Zaalouk, R. Khondoker, R. Marx, and K. Bayarou. Orchsec: An orchestrator-based architecture for enhancing network-security using network monitoring and SDN control functions. In *Network Operations and Management Symposium (NOMS)*, pages 1–9. IEEE, 2014.