FlowShark: Sampling for High Flow Visibility in SDNs

Sogand Sadrhaghighi* Mahdi Dolati† Majid Ghaderi* Ahmad Khonsari†
* University of Calgary, Calgary, Canada. Emails: {sogand.sadrhaghighi, mghaderi}@ucalgary.ca
† University of Tehran, Tehran, Iran. Emails: {mahdidolati, a_khonsari}@ut.ac.ir

***Abstract*—As the scale and speed of modern networks continue to increase, traffic sampling has become an indispensable tool in network management. While there exist a plethora of sampling solutions, they either provide limited flow visibility or have poor scalability in large networks. This paper presents the design and evaluation of FlowShark, a high-visibility per-flow sampling system for Software-Defined Networks (SDNs). The key idea in FlowShark is to separate sampling decisions on short and long flows, whereby sampling short flows is managed locally on edge switches, while a central controller optimizes sampling decisions on long flows. To this end, we formulate flow sampling as an optimization problem and design an online algorithm with a bounded competitive ratio to solve the problem efficiently. To show the feasibility of our design, we have implemented FlowShark in a small OpenFlow network using Mininet. We present experimental results of our Mininet implementation as well as performance benchmarks obtained from packet-level simulations in larger networks. Our experiments with a machine learning based Traffic Classifier application show up to 27% and 19% higher classification recall and precision, respectively, with FlowShark compared to existing sampling approaches.**

I. Introduction

Motivation. Network traffic monitoring is essential for network management tasks such as traffic engineering and anomaly detection [1], [2]. While high-level flow statistics are sufficient for some tasks (*e.g.*, traffic matrix estimation [3]), others require more detailed packet-level information, including packet payloads (*e.g.*, traffic classification [4]). Deploying passive monitoring equipment that captures every packet on a link provides highly accurate traffic information but scales poorly in large networks. As the scale and speed of modern networks continue to increase, more scalable and efficient solutions are needed for packet-level traffic monitoring. Many networks adopt solutions that use packet sampling in which network switches selectively capture a subset of packets that pass through them [5], [6]. Sampling, however, is an expensive operation for network switches to perform, given their limited processing capabilities. In particular, a typical switch can sample only a tiny fraction of the packets it sees, above which its performance begins to degrade [5]. Thus, when deploying sampling solutions, network operators use very low sampling rates to minimize any potential impact on network switches.

Legacy sampling solutions such as NetFlow [7] and sFlow [8] only support *per-port* sampling, in which a sampling rate is specified for each input port of the switch. As the packet arrival rate varies over time, the sampling rate varies accordingly. As a result, flows with higher packet rates are more likely to be sampled compared to flows with lower rates. A consequence of this is that short flows consisting of only a few packets may be entirely missed by the sampling system, resulting in *low visibility* of flows in the network. Clearly, such solutions are inadequate for network management tasks that require a minimum per-flow sampling rate in the network [9] such as anomaly detection tasks.

To address this limitation and enable *per-flow* sampling in a network, a number of solutions has been proposed [10]–[12]. In particular, the works [10] and [11] target legacy switches and, as such, require significant modifications of the packet processing pipeline of switches to identify and sample individual flows. The work [12], on the other hand, targets OpenFlow switches that natively support per-flow packet forwarding operations in Software-Defined Networks (SDNs). While the current OpenFlow specification (OpenFlow 1.5) does not support packet sampling, there are several proposals to add packet sampling extensions to OpenFlow [5], [13], [14]. As programmable switches such as those based on P4 [15] become mainstream, it is only reasonable to expect that per-flow sampling will be supported as a built-in feature on SDN switches. Indeed, the Switch Abstraction Interface (SAI) [16] developed by Open Compute Project and supported by all major switch vendors, already includes advanced features such as in-band telemetry, flow-based monitoring, and packet sampling in its Telemetry and Monitoring (TAM) specification [17].

Nevertheless, existing per-flow sampling solutions such as [10]–[12] *scale poorly* in large networks, as they rely on a network controller to determine a sampling rate for every flow on each switch along its path. In large networks such as datacenters, thousands of flows traverse each switch every second, with the majority of them being short flows that send only a few packets [18], [19]. For example, in Fig. 1, we have plotted the probability distribution of flow sizes in a datacenter using the publicly available web search workload [20]. As can be seen from the figure, close to 60% of flows send less than 100 KB of data. In a datacenter with 100 Gbps links, such flows finish in just 8 μs, which is less than the round-trip-time of the flows in the network [18]. By the time the network controller receives information about a short flow, determines switch sampling rates and instructs switches to sample the flow based on those rates, the flow may have already finished. Not only does such a design puts high control load on the controller, but also it may entirely miss some short flows, resulting in the same visibility problem as per-port solutions.

Our approach. Our approach is to take the best of both worlds: static pre-specified sampling for short flows but dynamic per-flow sampling for long flows. To this end, we present the design and evaluation of FlowShark, a scalable sampling system with high flow visibility for SDNs. The key

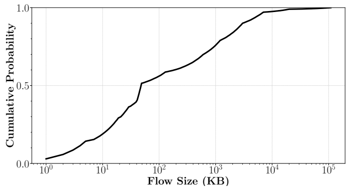


Fig. 1: Distribution of flow sizes in web search workload [20].

idea in FlowShark is the separation of sampling decisions on short and long flows. Specifically, sampling decisions on short flows are made locally on edge switches, while sampling decisions on long flows are made centrally by a controller application referred to as the sampling *orchestrator*. Whenever a new flow arrives to the network, it is assumed to be a short flow and will be sampled at a default rate on its corresponding edge switch. Once a flow is determined to be a long flow, it is handed off to the orchestrator which in turn determines on which switches and at what rates the flow should be sampled to satisfy a target sampling rate.

Recall that short flows constitute the majority of flows, while contributing only a small portion of the total traffic volume. In particular, in FlowShark, we use a threshold on the size of each flow to determine if the flow is short or long. Consequently, each short flow contributes only a negligible amount to the overall sampling load of the network. This means that the exact sampling locations and rates of short flows have a negligible impact on the efficiency of FlowShark. More importantly, sampling short flows locally allows FlowShark to fulfill two objectives: 1) improved scalability due to reduced load on the controller (*i.e.*, orchestrator), and 2) improved flow visibility by ensuring that every flow is sampled in the network regardless of its size. To optimally distribute sampling load among switches and, consequently, maximize the number of flows that are sampled at their target rate, FlowShark allows each flow to be partially sampled on multiple switches. The sampling decisions on long flows are made by the orchestrator. A major technical challenge in designing FlowShark is optimizing sampling decisions of the orchestrator without knowing the set of flows that will arrive in the network in the future. One of the main contributions of this work is the design and analysis of an *online* algorithm to make sampling decisions on long flows as they arrive in the network over time. This is in contrast to existing works [10], [11], [21], where sampling decisions are made in an *offline* manner, with an assumption that the set of flows in the network is fully known in advance.

Contributions. Our main contributions in this paper are:

- We present the design and evaluation of FlowShark, a scalable per-flow sampling system for SDNs. FlowShark includes simple and practical mechanisms for short and long flow detection as well as flow rate estimation.
- We formulate the problem of flow sampling as a linear program. Utilizing the primal-dual framework, we then design an online algorithm and analyze its competitive ratio and runtime complexity. We also design a simple greedy algorithm that has better runtime compared to the primal-dual-based algorithm but does not have a guaranteed worst case performance.
- We present a proof-of-concept Mininet implementation of FlowShark that includes lightweight mechanisms for long flow detection and traffic rate estimation. We also present packet-level simulation benchmarks in large networks to show the performance and scalability of FlowShark in both ISP and datacenter networks.
- We have implemented a realistic use case application for traffic classification based on machine learning on top of our Mininet prototype to demonstrate the utility of FlowShark in real-world applications.

Organization. The high-level design of FlowShark is presented in Section II. In Section III, we present our algorithms and their analysis. Evaluation results are presented in Section IV. Our concluding remarks are presented in Section VI.

II. System Design

The high-level architecture of FlowShark is depicted in Fig. 2. The main component of the design is the Orchestrator, which is implemented on top of the SDN controller. In the following, we first describe the workflow of the system and then provide more details about components of the architecture.

Workflow. There are two phases in FlowShark operation, namely the start-up phase and the sampling phase. In the startup phase, the orchestrator uses the controller's northbound interface to install default sampling rules for short flows. These rules are specified as wildcard rules with a default sampling rate and are installed only on edge switches. The destination of all sampling rules is the Collector server (*i.e.*, a load-balancer in front of storage servers), which is connected to an Analyzer application that implements the desired inference and management task. The orchestrator periodically polls edge switches for flow statistics (*e.g.*, OpenFlow counters). These statistics are forwarded to the Flow Classifier module, which determines if a flow is long, and if so, informs the orchestrator accordingly. Upon detection of a long flow, the orchestrator invokes the Rate Estimator module, which returns an estimate of the newly identified long flow traffic rate. The orchestrator also queries the controller to obtain the flow's path. It then invokes the optimizer module to compute a *sampling schedule* for the flow based on its path and the sampling schedules of existing flows. A sampling schedule for a flow determines the set of switches and their corresponding sampling rates for that flow. At this point, the orchestrator communicates with the SDN controller to install the newly computed sampling schedule on the relevant switches.

Orchestrator. The orchestrator is the brain of FlowShark. It is implemented as an application on the SDN controller and includes three modules, namely Flow Classifier, Rate Estimator and Optimizer. All communications between FlowShark and network switches go through the orchestrator. The orchestrator receives information about flows and their paths from the controller, communicates with its modules to build a sampling schedule for each long flow, and then, installs the schedule

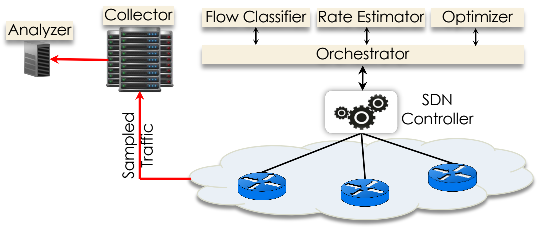


Fig. 2: High-level architecture of FlowShark.

with the help of the controller. The sampling schedules can be installed on OpenFlow switches via small OpenFlow protocol extensions [12], [13].

Flow Classifier. The flow classifier module of the orchestrator utilizes flow counters, polled periodically from edge switches by the orchestrator, to decide if a flow that has been continuing for a period of time is a long flow. Note that FlowShark does not need to know the total size of the flow in advance. Instead, it only checks if a flow is long or short. While a variety of techniques can be employed to distinguish between short and long flows, in FlowShark we adopt a simple technique based on a threshold on the amount of data transmitted so far by each flow (as in [22]–[24]). The threshold value can be tuned based on the sampling requirements of the analyzer application and network flow characteristics. If switch modifications are allowed, the flow classifier could be implemented on edge switches to avoid the overhead of polling flow counters.

Rate Estimator. To provide per-flow sampling at a pre-specified rate and compute an optimal sampling schedule for the flow, FlowShark estimates the sending rate of each long flow. Estimating traffic flow rates is a well-studied problem. For example, to estimate flow rates, the work [25] makes use of autoregressive models, while the works [26] and [27] apply machine learning techniques. In FlowShark, we use a feed forward neural network to estimate the traffic rate of each long flow based on the flow counters and traffic samples that have been collected while the flow was treated as a short flow. Details about the flow rate estimator implementation are presented in Section IV-A. As with the flow classifier module, the rate estimator could also be implemented on edge switches for improved efficiency, albeit with switch modification.

Optimizer. The optimizer module is in charge of computing sampling schedules for long flows, identified by the flow classifier module. When computing sampling schedules, the objective of the optimizer is to minimize the maximum sampling load among all switches. This objective enables FlowShark to optimally use switch sampling resources, which leads to higher flow visibility by allowing more flows to be sampled at the pre-specified target rate. Designing efficient algorithms for the optimizer is the main focus of the next section. Specifically, we focus on designing online algorithms that do not require knowledge about future flow arrivals in advance.

Discussion. OpenFlow switches have limited Ternary Content-Addressable Memory (TCAM) capacity for installing flow rules. One of the added benefits of using FlowShark compared to existing per-flow solutions, *e.g.*, [12], is that it only needs to install sampling rules for long flows that constitute a small portion of all flows in the network. As such, FlowShark reduces TCAM usage, compared to existing per-flow approaches.

III. Sampling Optimization

The optimizer module in FlowShark computes the sampling schedules by solving an optimization problem. Recall that a sampling schedule specifies the sampling locations (*i.e.*, switches) and associated sampling rates for a flow. In this section, we first formulate an offline optimization problem called Flow Sampling Rate Allocation (FSRA) for computing sampling schedules for a given set of flows. The objective of the FSRA problem is to minimize the maximum sampling load among all switches in the network.

The FSRA problem is a linear program (LP) that can be solved in polynomial time. In real-world applications, however, information about network flows is not known in advance, *i.e.*, the set of flows is unknown. Rather the optimizer module has to compute sampling schedules at runtime whenever a new (long) flow arrives. A naive solution is to recompute sampling schedules for all active flows every time a new long flow arrives and then re-install the new sampling schedules on network switches. Such a solution, however, is highly disruptive as it takes time for the orchestrator to work through the network controller and change the sampling rules on all switches on the path of active flows. Repeating this process every time a (long) flow arrives, which could be every few milli-seconds, is highly undesirable and may not be feasible in a large network.

To solve the FSRA problem in an online manner, we design two algorithms for computing sampling schedules as flows arrive sequentially over time. The first algorithm has a systematic design and is shown to have a bounded performance gap compared to the optimal offline algorithm with full knowledge about future flows. On the other hand, the second algorithm is a greedy strategy that is simple but does not provide guaranteed worst case performance.

Notation. Symbols denoting sets are typeset in calligraphic font, *e.g.*, set $\mathcal{F}$. The cardinality of set $\mathcal{F}$ is denoted by F (*i.e.*, typeset in regular font). Given variable H_z, for $z \in \mathcal{Z}$, we define $\overline{H}_z = \max_{z \in \mathcal{Z}} H_z$. The important notations used in this section are summarized in Table I.

A. Problem Formulation

Let $\mathcal{F}$ denote the set of (long) flows in the network to be sampled. The set of switches in the network is denoted by $\mathcal{S}$. The set of switches on the path of flow f and the set of flows that pass through switch s are denoted by $\mathcal{S}_f$ and $\mathcal{F}_s$, respectively. The traffic rate of flow f is denoted by r_f. Each flow can be sampled on multiple switches along its path. The goal is to sample every flow at a pre-specified aggregate rate ε, for $0 < \varepsilon < 1$. Let variable $\lambda_{f,s}$ denote the sampling rate of flow f on switch s. In order to satisfy the

TABLE I: Summary of Important Notations.

Notation	Description
$\mathcal{F}$	set of (long) flows to be sampled
$\mathcal{S}$	set of switches in the network
$\mathcal{F}_s$	set of flows that pass through switch s
$\mathcal{S}_f$	set of switches on the path of flow f
$\overline{F}_s$	max number of flows that pass through any switch
$\overline{S}_f$	max number of switches on the path of any flow
r_f	traffic rate of flow f
$\lambda_{f,s}$	sampling rate of flow f on switch s
$\boldsymbol{\lambda}$	vector of all flow sampling rates $\lambda_{f,s}$
Γ_s	sampling load of switch s
Γ	maximum switch sampling load
ε	pre-specified target sampling rate

Problem 1: Flow Sampling Rate Allocation (FSRA)

$$\begin{aligned}
&\min \Gamma && && \text{(5a)}\\
&\text{s.t. } \textstyle\sum_{f\in\mathcal{F}_s} r_f \cdot \lambda_{f,s} \le \Gamma, && \forall s \in \mathcal{S} && \text{(5b)}\\
&\quad \textstyle\sum_{s\in\mathcal{S}_f} \lambda_{f,s} \ge \varepsilon, && \forall f \in \mathcal{F} && \text{(5c)}\\
&\quad 0 \le \lambda_{f,s} \le 1, && \forall f \in \mathcal{F}, s \in \mathcal{S} && \text{(5d)}\\
&\quad \Gamma \ge 0. && && \text{(5e)}
\end{aligned}$$

sampling requirement of flow f, the following inequality must be satisfied:

$$1 - \textstyle\prod_{s\in\mathcal{S}_f}(1-\lambda_{f,s}) \ge \varepsilon\,. \tag{1}$$

Recall that to avoid degrading network switches forwarding performance, the target flow sampling rate is set to a very small value, *i.e.*, $\varepsilon \ll 1$, meaning that the sampling rate of a flow on each switch is also a very small value. As switches sample packets independently, applying the union bound to (1) yields the inequality $\sum_{s\in\mathcal{S}_f} \lambda_{f,s} \ge \varepsilon$, which is a linear constraint and is quite accurate in the regime of small sampling rates.

The LP formulation of the FSRA problem is presented in Problem 1. In this formulation, constraint (5b) enforces an upper bound on the sampling load of any switch in the network (denoted by Γ), which is then minimized across all switches in the objective of the problem. The product term $r_f\lambda_{f,s}$ in (5b) gives the sampling load of flow f on switch s. Notice that Γ is an optimization variable and is interpreted as the maximum sampling load among all switches in the network. By introducing Γ, we are able to transform the non-linear objective function $\min\max_{s\in\mathcal{S}}\Gamma_s$ to a linear objective function, where $\Gamma_s = \sum_{f\in\mathcal{S}_f} r_f \cdot \lambda_{f,s}$ gives the sampling load of switch s.

B. Online Flow Sampling Algorithm

The primal-dual technique has been successfully applied to design online algorithms for many offline problems formulated as LPs [28]. In this sub-section, we design an online algorithm for the FSRA problem, called Online Flow Sampling (OFS) algorithm, using the primal-dual framework developed in [29]. Moreover, using this framework, we show that OFS attains a polylogarithmic competitive ratio compared to the optimal offline algorithm with full knowledge of future flow arrivals.

Algorithm. The OFS algorithm relies on the dual of the FSRA problem. Thus, we first construct the dual of FSRA as:

$$\begin{aligned}
\textbf{Dual:}\quad &\max \varepsilon \textstyle\sum_{f\in\mathcal{F}} \beta_f && \text{(2a)}\\
&\text{s.t. } \beta_f - r_f \cdot \alpha_s \le 0, \quad \forall f \in \mathcal{F}, \forall s \in \mathcal{S}_f && \text{(2b)}\\
&\textstyle\sum_{s\in\mathcal{S}} \alpha_s \le 1, && \text{(2c)}\\
&\beta_f, \alpha_s \ge 0\,. && \text{(2d)}
\end{aligned}$$

where α_s and β_f are the dual variables associated with constraints (5b) and (5c), respectively. The OFS algorithm is outlined in Alg. 1. The algorithm takes as input a flow f and outputs a sampling schedule for the flow. At the start, the algorithm initializes the sampling rates of all switches along the flow's path (see line 4). The initial sampling rate of flow f on switch s is given by $\lambda^0_{f,s} = \frac{\varepsilon}{\rho\overline{F}_s\overline{S}_f}$, where $\overline{F}_s$, $\overline{S}_f$ and ρ denote the maximum number of flows that traverse any switch, the maximum number of switches on the path of any flow and the ratio of the maximum and minimum flow rate in the network, respectively. It is sufficient for the purpose of initialization to use upper bounds for these values. These initial values, however, may not satisfy constraint (5c). In that case, the algorithm multiplicatively updates the sampling rates of flow f on all switches along its path until constraint (5c) is satisfied, *i.e.*, the aggregate sampling rate of the flow on all switches along its path is greater than ε (see line 5). As the sampling rates are updated, the algorithm incurs an increase in the objective value, *i.e.*, increase in the maximum switch sampling load. To account for this increase, we define the following (differentiable) penalty function:

$$\Phi(\boldsymbol{\lambda}) = \log\big(\textstyle\sum_{s\in\mathcal{S}} \exp(\Gamma_s)\big), \tag{3}$$

where, $\boldsymbol{\lambda}$ is the sampling rate vector of all active flows in the network and Γ_s is the sampling load on switch s. Given this definition, it is straightforward to show that,

$$\Gamma \le \Phi(\boldsymbol{\lambda}) \le \Gamma + \log(S), \tag{4}$$

where, $\Gamma = \max_s \Gamma_s$ is the value of the objective of the online problem. The sampling rates of flow f on switches in $\mathcal{S}_f$ are increased multiplicatively proportional to the inverse of their penalty rate in order to force the algorithm to distribute the sampling load among different switches. Specifically, increasing the sampling rates on the switches that are already over-loaded will incur a significant penalty due to the exponential nature of the penalty function $\Phi(\boldsymbol{\lambda})$. After increasing the sampling rates, the algorithm updates the corresponding dual variable β_f. The specific expression used for updating the dual variable is designed to allow the analysis of the competitive ratio (see Theorem 1). The algorithm stops when constraint (5c) is satisfied.

Analysis. In the following, we will analyze the competitive ratio and runtime complexity of the OFS algorithm. The analysis considers changes in the values of the optimization variables in two consecutive iterations of the *while* loop. To distinguish between values of variables in different iterations, we use the notation X^l to refer to the value of variable X in iteration l of the loop, with X^0 denoting its initial value.

Definitions. To simplify mathematical expressions, define σ and μ as $\sigma = \exp(1+\varepsilon)\overline{S}_f \log(\rho\overline{S}_f\overline{F}_s)$ and $\mu = 1+\frac{1}{3\log(e\overline{S}_f)}$.

Alg. 1: Online Flow Sampling (OFS)

```
procedure OFS ($f$)
$\beta_f \leftarrow 0$
for $s \in \mathcal{S}_f$ do
    $\lambda_{f,s} \leftarrow \frac{\varepsilon}{\rho \overline{F}_s \overline{S}_f}$
while Constraint (5c) is not satisfied do
    Calculate the penalty rate for every switch $s \in \mathcal{S}_f$:
        $\nabla_{f,s}(\boldsymbol{\lambda}) \leftarrow \frac{\partial \Phi(\boldsymbol{\lambda})}{\partial \lambda_{f,s}} = \frac{r_f \cdot \exp(\Gamma_s)}{\sum_{s' \in \mathcal{S}} \exp(\Gamma_{s'})}$    (6a)
    Calculate the normalized minimum penalty rate $\Pi_f(\boldsymbol{\lambda})$:
        $\Pi_f(\boldsymbol{\lambda}) \leftarrow \frac{1}{3\log(e\overline{S}_f)} \min_{s \in \mathcal{S}_f} \nabla_{f,s}(\boldsymbol{\lambda})$    (6b)
    Increase $\lambda_{f,s}$ for every switch $s \in \mathcal{S}_f$:
        $\lambda_{f,s} \leftarrow \lambda_{f,s}\Big(1 + \frac{\Pi_f(\boldsymbol{\lambda})}{\nabla_{f,s}(\boldsymbol{\lambda})}\Big)$    (6c)
    Increase the corresponding dual variable $\beta_f$:
        $\beta_f \leftarrow \beta_f + \exp(\varepsilon) \cdot \Pi_f(\boldsymbol{\lambda})$    (6d)
return $\lambda_{f,s}$
```

Notice that $\sigma = \mathcal{O}(S \log(SF))$, while $\mu = \mathcal{O}(1)$.

Lemma 1 (**Convergence**). *Algorithm OFS converges to a feasible solution in $\mathcal{O}\big(\overline{S}_f \log(e\overline{S}_f)\log(\rho\overline{F}_s\overline{S}_f)\big)$ iterations.*

Proof: There are S_f switches on the path of flow f and, thus, S_f sampling variables. The loop terminates when $\sum_{s\in\mathcal{S}_f} \lambda^l_{f,s} \geq \varepsilon$ in some iteration l. From (6c), it is obtained that, for every switch s, $\lambda^l_{f,s} < \lambda^{l+1}_{f,s} \leq \mu \cdot \lambda^l_{f,s}$. This implies that the values of the sampling variables for each switch form a strictly increasing sequence. As such, the loop will eventually terminate after a finite number of iterations. Consider the switch that minimizes (6b) in some iteration of the loop. For this switch, its sampling rate increases exactly by a factor of μ In the worst case, it takes S_f iterations to increase each sampling variable by at least μ. After L iterations of the loop, we have $\lambda^L_{f,s} \geq \lambda^0 \mu^{(L/S_f)}$. Solving for $\lambda^0\mu^{(L/S_f)} \leq \varepsilon$, yields the expression $L \leq S_f \log(\varepsilon/\lambda^0)/\log(\mu)$. Applying the inequality $1 + a \geq \exp(\frac{a}{e})$, for $0 \leq a \leq 1$, it is obtained that $\log(\mu) \geq \frac{1}{3e\log(e\overline{S}_f)}$. Using this, we have that $L \leq 3eS_f \log(e\overline{S}_f)\log(\rho\overline{F}_s\overline{S}_f)$. Taking the maximum of this bound over all flows establishes the lemma. ■

Lemma 2 (**Dual Feasibility**). *Define $\nu = \log(eS) + \Gamma$. Then, $\frac{\beta_f}{\sigma\nu}$ and $\frac{\alpha_s}{\nu}$ are feasible solutions for the dual problem.*

Proof: From (6d), we know that $\beta_f = \sum_l \exp(\varepsilon)\Pi_f(\boldsymbol{\lambda}^l)$. As per (6b), we can drive the following relations,

$$\begin{aligned}\textstyle\sum_l \Pi_f(\boldsymbol{\lambda}^l) &= \tfrac{1}{3\log(e\overline{S}_f)} \textstyle\sum_l \min_{s\in\mathcal{S}_f} \nabla_{f,s}(\boldsymbol{\lambda}^l),\\ &\leq \tfrac{1}{3\log(e\overline{S}_f)} \max_l \big(\min_{s\in\mathcal{S}_f} \nabla_{f,s}(\boldsymbol{\lambda}^l)\big) \cdot L, \quad (7)\end{aligned}$$

where, L is the number of iterations of the loop. Applying Lemma 1 to the number of loop iterations, it is obtained that,

$$\beta_f \leq \sigma \max_l \min_{s\in\mathcal{S}_f} \nabla_{f,s}(\boldsymbol{\lambda}^l)\,. \quad (8)$$

Using the above inequality, if we set the dual variable α_s so that $r_f \cdot \alpha_s$ is at least $\max_l \min_{s\in\mathcal{S}_f} \nabla_{f,s}(\boldsymbol{\lambda}^l)$ then the dual constraint (2b) is violated by a factor of at most σ. As such, $\frac{\beta_f}{\sigma}$ is a feasible dual solution. We also set the value of dual variable α_s as $\alpha_s = \max_l \min_{s\in\mathcal{S}_f} \frac{\nabla_{f,s}(\boldsymbol{\lambda}^l)}{r_f}$. Since the dual problem is a maximization problem and $\frac{\beta_f}{\sigma}$ is a feasible dual solution, we have that $\frac{\beta_f}{\sigma\nu}$ and $\frac{\alpha_s}{\nu}$ are feasible as well. ■

Theorem 1. *Algorithm OFS is $8\sigma\log(eS)$-competitive.*

Proof: Consider the set of flows $\mathcal{F}$. Let OPT denote the objective value of the optimal offline solution for this set of flows. We assume that the flow rates are scaled, otherwise the doubling procedure can be employed [30], so that the inequality $\frac{1}{4\sigma} \leq \text{OPT} \leq \frac{1}{2\sigma}$ holds. To establish the theorem, we have to prove that $\Gamma \leq 8\sigma\log(eS)\text{OPT}$, where Γ is the value of the objective achieved by the OFS algorithm for the flow set $\mathcal{F}$.

First, we establish a lower bound on OPT. For any flow f, its sampling rate on at least one of the switches along its path should be greater than or equal to $\frac{\varepsilon}{S_f}$. Considering the flow with the longest path in the network, this provides a lower bound on OPT as $\text{OPT} \geq \frac{\varepsilon}{\overline{S}_f}\min_{f\in\mathcal{F}} r_f$. From this, the following is established,

$$\begin{aligned}\text{OPT} &\geq \tfrac{\varepsilon}{\overline{S}_f}\min_{f\in\mathcal{F}} r_f = \tfrac{\varepsilon}{\overline{S}_f}\min_{f\in\mathcal{F}} r_f \cdot \tfrac{\max_{f\in\mathcal{F}} r_f}{\max_{f\in\mathcal{F}} r_f},\\ &= \tfrac{\varepsilon}{\overline{S}_f}\tfrac{\max_{f\in\mathcal{F}} r_f}{\rho} = \tfrac{\varepsilon}{\rho\overline{S}_f}\max_{f\in\mathcal{F}} r_f\,. \quad (9)\end{aligned}$$

Next, consider the behavior of the online algorithm. The online algorithm assigns initial values to sampling variables of a flow at the arrival time of the flow. Then, it goes through the loop described in Alg. 1 to gradually increase the initial rates to their final values. Let Γ^0 denote the value of the online objective assuming that all flows are sampled using the initial sampling rates. At this step, the following relation holds:

$$\begin{aligned}\Gamma^0 &= \max_{s\in\mathcal{S}} \textstyle\sum_{f\in\mathcal{F}_s} r_f \cdot \lambda^0_{f,s} \leq \overline{F}_s \cdot \lambda^0_{f,s}\max_{f\in\mathcal{F}} r_f,\\ &= \tfrac{\varepsilon}{\rho\overline{S}_f}\max_{f\in\mathcal{F}} r_f \leq \text{OPT}\,. \quad (10)\end{aligned}$$

Recall that $\Gamma \leq \Phi(\boldsymbol{\lambda}) \leq \log(S) + \Gamma$, for any arbitrary vector of sampling rates. Therefore, using the initial sampling rates vector $\boldsymbol{\lambda}^0$, we have $\Phi(\boldsymbol{\lambda}^0) \leq \log(S) + \Gamma^0$. Putting these together, it is obtained that,

$$\Phi(\boldsymbol{\lambda}^0) \leq \text{OPT} + \log(S)\,. \quad (11)$$

For each flow, in each iteration of the algorithm, the increase in penalty $\Phi(\boldsymbol{\lambda})$ is bounded by the increase in the dual's objective. When the while loop terminates, the total increase in the dual's objective due to flow f is given by $\varepsilon\beta_f$. Thus, when the last flow arrives and is processed by the algorithm, the total increase in $\Phi(\boldsymbol{\lambda})$ over all flows after their initialization step is bounded by $\sum_{f\in\mathcal{F}}\varepsilon\beta_f$. Combining this with the upper bound for the initialization step results in the following inequality:

$$\begin{aligned}\Gamma \leq \Phi(\boldsymbol{\lambda}) &\leq \textstyle\sum_{f\in\mathcal{F}}\varepsilon\beta_f + \Phi(\boldsymbol{\lambda}^0),\\ &\leq \textstyle\sum_{f\in\mathcal{F}}\varepsilon\beta_f + \log(S) + \text{OPT}\,. \quad (12)\end{aligned}$$

From Lemma 2, we know that $\frac{\beta_f}{\sigma\nu}$ and $\frac{\alpha_s}{\nu}$ are feasible dual solutions. Using the weak duality, any feasible dual solution is a lower bound on the primal's objective. As a result, the following relation must hold,

$$\textstyle\sum_{f\in\mathcal{F}}\varepsilon\big(\frac{\beta_f}{\sigma\nu}\big) \leq \text{OPT}\,. \quad (13)$$

By substituting in (12), the following relation is established:

$$\Gamma \leq (\sigma\nu + 1)\mathrm{OPT} + \log(S)\,. \tag{14}$$

Noting the upper bound $\mathrm{OPT} \leq \frac{1}{2\sigma}$ and substituting the value of ν, we obtain that,

$$\Gamma \leq \frac{\Gamma + \log(eS)}{2} + \frac{1}{2\sigma} + \log(S), \tag{15}$$

which, in turn yields the following inequality,

$$\Gamma \leq \log(eS) + \frac{1}{\sigma} + 2\log(S) \leq 3\log(eS)\,. \tag{16}$$

Using the above inequality and the definition of ν, we obtain that $\nu \leq 4\log(eS)$. The following relation is obtained by substituting for ν in (14),

$$\Gamma \leq 4\sigma\log(eS)\mathrm{OPT} + \mathrm{OPT} + \log(S)\,. \tag{17}$$

The inequality $\log(S) \leq \log(S) \cdot 4\sigma\mathrm{OPT}$ is always true given the lower bound $\mathrm{OPT} \geq \frac{1}{4\sigma}$. By substituting in the above inequality the theorem is established as,

$$\begin{aligned}\Gamma &\leq 4\sigma\log(eS)\mathrm{OPT} + \mathrm{OPT}(1 + 4\sigma\log(S)),\\ &\leq 8\sigma\log(eS)\mathrm{OPT}\,.\end{aligned} \tag{18}$$

■

Runtime Analysis. The runtime of the algorithm is dominated by the *while* loop. As shown in Lemma 1, the number of iterations in the while loop is $\mathcal{O}\big(\overline{S}_f \log(e\overline{S}_f)\log(\rho\overline{F}_s\overline{S}_f)\big)$. Additionally, in each iteration, the algorithm iterates over all switches to compute their sampling load (using (6a)), which in the worst case, takes $\mathcal{O}(\overline{F}_s \cdot \overline{S}_f)$. Therefore, the runtime of the algorithm is $\mathcal{O}(\log(e\overline{S}_f) \cdot \log(\rho\overline{F}_s \cdot \overline{S}_f) \cdot \overline{F}_s \cdot \overline{S}_f^2)$.

C. Greedy Flow Sampling Algorithm

The OFS algorithm follows the primal-dual framework, which allows us to analytically derive its competitive ratio. The advantage of this approach is that the gap between the performance of OFS and the offline optimal is guaranteed in the worst case. In this sub-section, we design a greedy online algorithm for the the FSRA problem called Greedy Flow Sampling (GFS) algorithm, which does not provide a guaranteed worst case performance, but is much simpler to implement and runs faster than OFS. Our experimental results in Seciton IV show that it also performs remarkably close to OFS in the range of scenarios we evaluated.

Algorithm. The GFS algorithm is presented in Alg. 2. Upon the arrival of a flow, if there exists a switch on the flow's path with no sampling load, the flow is fully sampled on that switch (see lines 3-4). If such a switch does not exist, then the algorithm decides on the sampling rates of the flow on different switches along the flow's path. Specifically, it starts by finding the switch that has the maximum sampling load along the flow's path. The algorithm then uses the sampling load of this switch to calculate a *share* for every switch, accounting for their current sampling loads (see lines 7-9). The calculated shares are used to distribute the target sampling rate of the flow among all the switches (see line 11).

Runtime Analysis. The runtime of the algorithm is dominated by the first *for* loop. Each iteration of this loop takes $\mathcal{O}(\overline{F}_s)$ time. Therefore, the algorithm runs in $\mathcal{O}(\overline{S}_f \cdot \overline{F}_s)$ time.

Alg. 2: Greedy Flow Sampling (GFS)

```
1  procedure GFS ($f$)
2  weight ← $0$
3  if $s \in \mathcal{S}_f$ and $\Gamma_s == 0$ then
4      $\lambda_{f,s} \leftarrow \varepsilon$
5  else
6      foreach $s \in S_f$ do
7          $w_s \leftarrow \frac{\max_{s'} \Gamma_{s'}}{\Gamma_s}$
8          $weight \leftarrow weight + w_s$
9      $share \leftarrow \frac{\varepsilon}{weight}$
10     foreach $s \in \mathcal{S}_f$ do
11         $\lambda_{f,s} \leftarrow share \cdot w_s$
12 return $\lambda_{f,s}$
```

IV. Evaluations

In this section, we study the performance and behavior of FlowShark using Mininet experiments and simulations.

Methodology. Mininet experiments consider a small network and show the feasibility of implementing FlowShark in OpenFlow networks. Simulation experiments, on the other hand, consider larger networks to show the scalability of FlowShark. We present the following evaluation results:

A. System Performance: These experiments are conducted in Mininet to study system-level performance of FlowShark including flow visibility and sampling load.

B. Simulation Benchmarks: Simulations are used to understand the low-level behaviour of FlowShark, namely the impact of flow rate estimation errors and the empirical competitive ratio of OFS.

C. Application Performance: These experiments are conducted in Mininet to show the performance of FlowShark when used in conjunction with a network management application, namely a *Traffic Classifier*.

Implementation. We used Mininet to implement FlowShark in an OpenFlow network. We deploy ONOS [31] as the SDN controller. Routing flow rules are installed using proactive forwarding to avoid varying routes for the same flow among different experiment runs. We utilize optional flow tables in OpenFlow switches to install per-flow sampling rules. Specifically, in OpenFlow, a pipeline of multiple flow tables is used to decouple different network functions [32]. Pipeline processing starts with the forwarding table, in which the matching field specifies a set of instructions, directing the flow to the sampling table. When pipeline processing stops, the packet is processed with its associated action set, *i.e.*, forward and sample. Finally, the rate estimator is implemented as a feed forward neural network (NN) using the MLPRegressor module of the scikit Python library. The NN is trained on features that can be easily obtained from the OpenFlow switches, namely IP protocol, source and destination ports, along with some additional flow-based features like packet sizes. These features are easily obtainable from OpenFlow switches early in the flow's life-cycle. The NN has one hidden layer with 100 neurons. ReLU is used as the activation function. The model is trained using Adam optimizer on the datacenter packet traces available in [33] (studied in [34]).

A. System Performance

Setup. These experiments consider a 2-pod FatTree topology consisting of 8 hosts and 10 OpenFlow switches. Following the switch specification [35], the sampling capacity of each switch is set to 50 packets per second (pps). The bandwidth of each link in the network is set to 20 Mbps. The target sampling rate for each flow is set to $\varepsilon = 0.1$. Note that we set the sampling capacity of switches to a low number to amplify the effect of sampling on flow visibility. In practice, sampling capacity, number of flows and link capacities are orders of magnitude higher than what we have considered here. However, to allow Mininet experiments to finish in a reasonable time, it is necessary to scale such network features. To generate a flow, we choose a random source and destination. The traffic rate between each source and destination is chosen based on the widely used datacenter web server workload [36]. The packet size is set to 1500 Bytes. The reported results are averaged over 5 experiment runs.

Metrics. We report the following performance metrics:

- *Missed Flows:* The percentage of flows for which no packet is sampled. A higher percentage of missed flows indicates lower flow visibility.
- *Sampled Flow Rate:* The actual measured sampling rate of a flow. Ideally, the measured sampled flow rate should closely match the target rate $\varepsilon = 0.1$.
- *Network Sampling Load:* The total sampled traffic on all switches in the network normalized by the switch sampling capacity. We present this metric using the spread and centers of the total measured sampled traffic.
- *Maximum Switch Sampling Load:* The sampling load of the switch with the highest sampling load in the network normalized by the switch sampling capacity. For the purposes of measuring this metric, we assume that there is no limit on the sampling capacity of switches.

Algorithms. In addition to OFS and GFS, we also implemented the following algorithms for comparison:

- *Uniform Packet Sampling* (UPS): Each switch samples every arriving packet with probability ε. UPS is a representative of per-port sampling solutions.
- *Uniform Flow Sampling* (UFS): Each switch samples every flow with rate ε. Once a switch has captured ε fraction of a flow, it stops sampling that flow. To achieve this, UPS assumes a priori knowledge about flow rates. UFS represents uniform per-flow sampling solutions.

Visibility Analysis. Fig. 3 presents the results of visibility analysis. As can be seen in Fig. 3(a), OFS achieves the lowest percentage of missed flows. A back-of-the-envelope calculation shows that the total sampling capacity of the network is sufficient to fully sample only 38 flows. With routing restrictions, even fewer flows can be fully sampled in the network, particularly when there is a small number of flows to choose from. To understand the poor performance of UPS and UFS, in Fig. 3(b), we have plotted the CDF of measured sampled flow rates. One can observe that when using UPS and UFS, 25% of the flows are sampled at a rate higher than 2x of the target sampling rate of $\varepsilon = 0.1$. This shows a bias towards some of the flows, which consequently results in nearly 40% of the flows being entirely missed.

Load Analysis. Fig. 4 presents the results of sampling load analysis. From Fig. 4(a), we observe that OFS and GFS achieve the lowest maximum switch sampling load. This is indeed one reason for higher flow visibility under these algorithms. Not only OFS and GFS avoid over-sampling flows, but also distribute sampling load more evenly among switches, avoiding sampling bottlenecks. These two characteristics result in higher flow visibility, as observed in Fig. 3. In Fig. 4(b), we show the network sampling load distribution. Besides the expected higher load of UPS and UFS, an interesting observation is with respect to the sampling load of GFS compared to OFS. We can see that in OFS, the first quartile and the minimum sampling load are closer to each other compared to GFS. Specifically, OFS incurs 25% lower sampling load. In summary, not only OFS provides higher flow visibility, it also imposes a lower sampling load on the network.

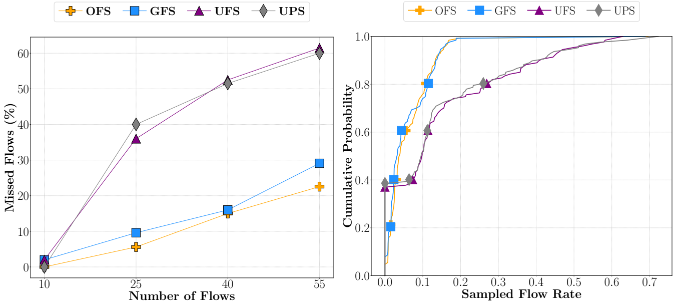
B. Simulation Benchmarks

Setup. We implemented the following network topologies in our simulations: 1) USA ISP topology, chosen from the topology Zoo dataset [37]. It contains 24 switches and 24 hosts. 2) 12-pod FatTree topology with 180 switches and 72 hosts. Note that a FatTree topology of this size is larger or equal to the ones commonly used in the literature [38]. We generate traffic flows between randomly chosen source-destination pairs, with the flow rates randomly selected from a uniform distribution in the interval $[2, 6]$ Mbps. Additionally, the target sampling rate for each flow is set to $\varepsilon = 0.002$. The reported results are averaged over 5 simulation runs. Simulations are conducted using a custom-built packet-level simulator implemented in Python programming language.

Experiments. We report two sets of experiments:

- *Empirical Competitive Ratio*: We compare the maximum switch sampling load achieved using OFS and the optimal offline solution (OPT). OPT is obtained by solving Problem 1 using Gurobi optimizer [39], assuming full knowledge of future flow arrivals and their exact traffic rates.
- *Rate Estimation Errors*: We first evaluate the sensitivity of OFS to flow rate estimation errors. Then, we measure the effect of estimation errors on the maximum switch load.

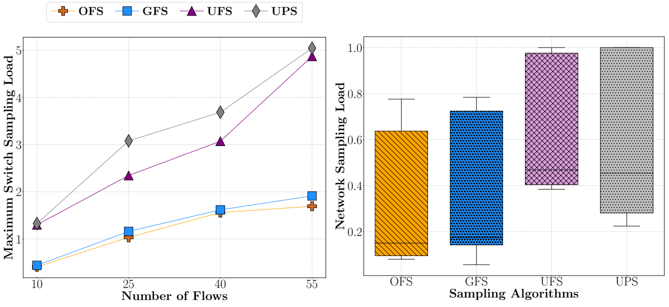
Empirical Competitive Ratio. Fig. 5(a) and Fig. 5(b) show the maximum switch load under OFS and OPT on the USA and FatTree topology, respectively. The maximum switch load of OFS is on average 1.7 and 2.8 times of that under OPT on the USA and FatTree topology, which are substantially lower than the corresponding theoretical competitive ratios. It is worth emphasizing that OPT is an unrealistic algorithm: it is an oracle with 20/20 vision into the future. Thus, it is no surprise that it outperforms OFS, which exists in real world.



(a) Percentage of missed flows.

(b) CDF of sampled flow rates (25 flows).

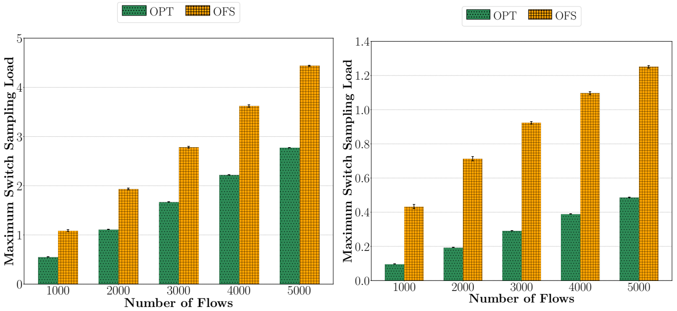
Fig. 3: Flow visibility based on sampled flow rates.



(a) Maximum switch sampling load.

(b) Network sampling load (25 flows).

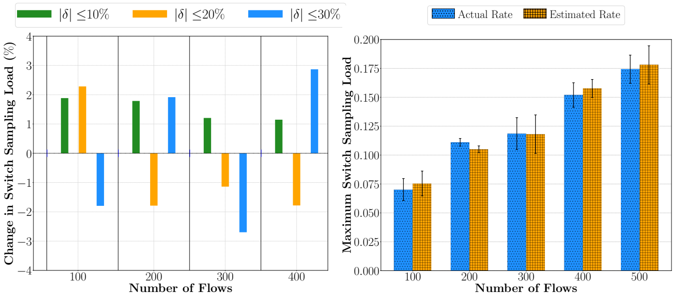
Fig. 4: Switch and network sampling load.



(a) USA topology.

(b) FatTree topology.

Fig. 5: Comparison with the optimal offline algorithm.



(a) Sensitivity to flow rate estimation errors.

(b) Effect of rate estimation.

Fig. 6: Effect of flow rate estimation errors.

Rate Estimation Errors. First, we consider a range of estimation errors, denoted by δ, from 10% to 30% of the actual flow rates. We generate a random estimation error within the given range for each flow and add it to the actual flow rate to estimate its rate. This estimated rate (which includes an error) is then fed to OFS for sampling optimization. The results are presented in Fig. 6(a). Each bar on the plot represents the percentage of change in maximum switch sampling load with rate estimation errors with respect to the case when actual flow rates are used in OFS. From the figure, we observe that as much as 30% estimation error has negligible effect. Next, we study OFS performance when using FlowShark NN-based rate estimator versus using the actual rates. The results are presented in Fig. 6(b) for the FatTree topology (results for the USA topology were similar). We observe that the difference between the maximum switch load with the actual and estimated rates is negligible (*e.g.*, only 4% difference in the worst case).

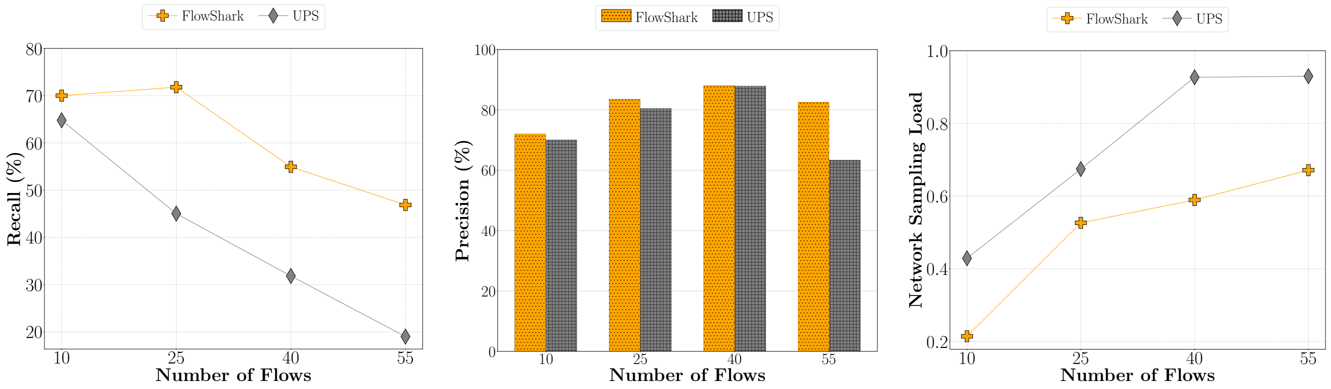
C. Application Performance

Setup. We implemented a Traffic Classifier application on top of FlowShark implementation in Mininet. The network setup is similar to the one used in sub-Section IV-A. To generate traffic, we use the dataset available on [37], and studied in [40]. In particular, we target traffic from the following categories: Google, Amazon, Youtube, SSL and HTTP. To generate traffic, we randomly select source and destination hosts and use the information provided in the dataset, *e.g.*, source and destination ports and packet lengths. Additionally, FlowShark uses the threshold of 10 KB to distinguish short flows. The sampled traffic is passed to the Traffic Classifier application. The classifier application functions similar to existing works, *e.g.*, [4]. In our implementation, we use the C-Support Vector Classifier (SVC) from the scikit-learn Python library [41]. The classifier is initialized with regularization parameter $C = 5$ and uses "rbf" kernel. The features used for classification are the packet header fields (*i.e.*, source and destination port numbers, packet length, protocol) and first 10 Bytes of the packet payload. The results presented are averaged over 5 experiment runs.

Metrics. In addition to the *Network Sampling Load*, we report the following performance metrics:

- *Recall*: Number of correctly classified flows divided by the total number of generated flows.
- *Precision*: The number of correctly classified flows divided by the total number of sampled flows (excluding the missed flows).

Results. Figs. 7(a) and 7(b) present the recall and precision metrics, respectively. From Fig. 7(a), it can be observed that by increasing the number of flows, the classification recall follows a downward trend under both FlowShark and UPS. This is expected due to limited sampling capacity of switches. However, what is important is that the recall performance under FlowShark does not degrade as much as under UPS. In particular, under FlowShark, the classifier achieves up to 27% higher recall than UPS. From Fig. 7(b), we can see that as the number of flows increases, which is when the mechanics of the sampling algorithm come to play, the classifier's precision substantially drops under UPS compared to FlowShark. For example, it drops by 24% when increasing the number of flows from 40 to 55. The higher recall and precision imply that, when



(a) Traffic classification recall.

(b) Traffic classification precision.

(c) Network sampling load.

Fig. 7: Mininet experiment results for Traffic Classifier application performance.

used with the same classification model, FlowShark achieves higher flow visibility compared to UPS. The network sampling load is depicted in Fig. 7(c). The important observation in this figure is the plateauing behavior of UPS when the number of flows reaches 40. This behavior suggests that due to sampling more packets of the same flows, UPS exhausts the network sampling capacity, reaching the maximum sampling load with fewer flows. Furthermore, by comparing the recall (Fig. 7(a)) with the sampling load, one can conclude that the increased sampling load does not necessarily result in higher flow visibility. A per-flow sampling solution such as FlowShark not only improves application performance, but also lowers the network sampling load.

V. Related Works

Per-Port Sampling. Traditionally, NetFlow [7], IPFIX [42] and sFlow [8] have been used for traffic sampling. To minimize sampling load on switches, the authors in [43] consider distributing sampling load among different switches. Their approach, unlike FlowShark, requires advanced knowledge of all flows and is not a per-flow sampling framework. The works [44] and [9] focus on determining flow sampling locations. In particular, [9] chooses switches with the goal of maximizing a utility function, while [44] makes use of the centrality measure in graph theory. The works in this category consider fixed predefined sampling rates per switch interface, which as discussed earlier, can result in low flow visibility.

Per-Flow Sampling. Several works propose to use sampling for per-flow statistics collection. Examples of such works are [45] and [46], both of which employ sketches to probabilistically collect per-flow statistics. Implementing per-flow sampling to collect packet-level information is studied in [47] and [13]. Specifically, [47] proposes a per-flow sampling framework for legacy switches, while [13] extends the OpenFlow specification to support packet sampling in SDNs. These works, however, do not consider sampling coordination among switches. The works [10], [11], [21], on the other hand, distribute the sampling load of a flow among different switches, but they assume full knowledge about the set of flows in advance. Per-flow sampling in SDNs is considered in [12], where a controller determines the sampling rate of each flow. The works in this category suffer from poor scalability in large networks as they rely on central decision making for all flows (*i.e.*, short and long flows) in the network.

Time-Driven Sampling. In time-derive sampling, switches sample consecutive packets at specific time intervals, rather than randomly sampling packets. Examples of such solutions are presented in [48] and [49]. In particular, [48] proposes a time-driven sampling framework for SDNs, in which each switch forwards consecutive packets of a flow to the controller in specific intervals. To minimize the sampling load, the authors in [49] choose the k most influential switches based on the spatial-temporal factors in the network, and assign sampling intervals only to those switches. Their approach, however requires a priori knowledge of the set of flows and time synchronization between different switches.

Application-Specific Sampling. Several works study designing sampling solutions to increase the accuracy of specific applications, *e.g.*, security analysis and traffic classification. Sampling for botnet detection and intrusion detection are considered in [50] and [51], respectively. Per-port sampling for traffic classification is studied in [52] and [53]. These works only consider per-port or time-driven sampling. On the other hand, FlowShark is a per-flow sampling system and is not limited to a specific application.

VI. Conclusion

This work presented a traffic sampling system called FlowShark to address the low flow visibility in existing sampling solutions for large-scale networks. By delegating short flow sampling decisions to edge switches, FlowShark optimizes sampling for long flows that can afford the latency of a centralized control loop. To show the feasibility of FlowShark in real-world networks, we provided a detailed description of our implementation in Mininet. We then used the implementation in conjunction with simulations to evaluate the performance of FlowShark and demonstrate its advantages over existing solutions. A worthwhile extension of FlowShark is to consider different sampling rates for different flows. Furthermore, it would be interesting to consider implementing FlowShark on P4 switches to allow partial packet sampling.

References

[1] P.-W. Tsai, C.-W. Tsai, C.-W. Hsu *et al.*, "Network monitoring in software-defined networking: A review," *IEEE Systems Journal*, vol. 12, no. 4, 2018.

[2] A. Zaalouk, R. Khondoker, R. Marx *et al.*, "OrchSec: An orchestrator-based architecture for enhancing network security using network monitoring and SDN control functions," in *Proc. IEEE NOMS*, May 2014.

[3] Y. Tian, W. Chen, and C.-T. Lea, "An SDN-based traffic matrix estimation framework," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 4, 2018.

[4] J. Zhang, Y. Xiang, W. Zhou *et al.*, "Unsupervised traffic classification using flow statistical properties and IP packet payload," *Elsevier Journal of Computer and System Sciences*, vol. 79, no. 5, 2013.

[5] J. Suh, T. T. Kwon, C. Dixon *et al.*, "Opensample: A low-latency, sampling-based measurement platform for commodity SDN," in *Proc. IEEE ICDCS*, Jun 2014.

[6] Y. Li, R. Miao, C. Kim *et al.*, "Flowradar: A better NetFlow for datacenters," in *Proc. USENIX NSDI*, Mar 2016.

[7] B. Claise, "Cisco systems NetFlow services export v9," accessed Jul. 29, 2021. [Online]. Available: https://www.ietf.org/rfc/rfc3954.txt

[8] sFlow, "Making the network visible," accessed Jul. 29, 2021. [Online]. Available: https://sflow.org/

[9] G. R. Cantieni, G. Iannaccone, C. Barakat *et al.*, "Reformulating the monitor placement problem: Optimal network-wide sampling," in *Proc. ACM CoNEXT*, Dec 2006.

[10] V. Sekar, M. K. Reiter, W. Willinger *et al.*, "cSamp: A system for network-wide flow monitoring," in *Proc. USENIX NSDI*, Apr 2008.

[11] V. Sekar, M. K. Reiter, and H. Zhang, "Revisiting the case for a minimalist approach for network flow monitoring," in *Proc. ACM SIGCOMM*, Sep 2010.

[12] R. Cohen and E. Moroshko, "Sampling-on-demand in SDN," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, 2018.

[13] S. Shirali-Shahreza and Y. Ganjali, "Flexam: Flexible sampling extension for monitoring and security applications in OpenFlow," in *Proc. ACM SIGCOMM HotSDN*, Aug 2013.

[14] J. Suárez-Varela and P. Barlet-Ros, "Towards a NetFlow implementation for OpenFlow software-defined networks," in *Proc. IEEE ITC*, Oct 2017.

[15] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[16] O. C. Project, "SAI-Switch abstraction interface," accessed Jul. 29, 2021. [Online]. Available: https://github.com/opencomputeproject/SAI

[17] ——, "Telemetry and monitoring (TAM) specification," accessed Jul. 29, 2021. [Online]. Available: https://github.com/opencomputeproject/SAI/tree/master/doc/TAM

[18] B. Montazeri, Y. Li, M. Alizadeh *et al.*, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. ACM SIGCOMM*, Aug 2018.

[19] S. Hu, W. Bai, G. Zeng *et al.*, "Aeolus: A building block for proactive transport in datacenters," in *Proc. ACM SIGCOMM*, Jul 2020.

[20] M. Alizadeh, A. Greenberg, D. A. Maltz *et al.*, "Datacenter TCP (DCTCP)," in *Proc. ACM SIGCOMM*, Aug 2010.

[21] C.-W. Chang, G. Huang, B. Lin *et al.*, "Leisure: A framework for load-balanced network-wide traffic measurement," in *Proc. ACM/IEEE ANCS*, Oct 2011.

[22] M. Al-Fares, S. Radhakrishnan, B. Raghavan *et al.*, "Hedera: Dynamic flow scheduling for datacenter networks." in *Proc. USENIX NSDI*, Apr 2010.

[23] G. Porter, R. Strong, N. Farrington *et al.*, "Integrating microsecond circuit switching into the data center," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, 2013.

[24] W. M. Mellette, R. Das, Y. Guo *et al.*, "Expanding across time to deliver bandwidth efficiency and low latency," in *Proc. USENIX NSDI*, Feb 2020.

[25] Q. He, X. Wang, and M. Huang, "Openflow-based low-overhead and high-accuracy SDN measurement framework," *Wiley Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 2, 2018.

[26] D. Andreoletti, S. Troia, F. Musumeci *et al.*, "Network traffic prediction based on diffusion convolutional recurrent neural networks," in *Proc. IEEE INFOCOM WKSHPS*, May 2019.

[27] S.-C. Chao, K. C.-J. Lin, and M.-S. Chen, "Flow classification for software-defined data centers using stream mining," *IEEE Trans. Services Comput.*, vol. 12, no. 1, 2016.

[28] N. Alon, B. Awerbuch, Y. Azar *et al.*, "A general approach to online network optimization problems," *ACM TALG*, vol. 2, no. 4, 2006.

[29] Y. Azar, U. Bhaskar, L. Fleischer *et al.*, "Online mixed packing and covering," in *Proc. ACM/SIAM SODA*, Jan 2013.

[30] L. Besson and E. Kaufmann, "What doubling tricks can and can't do for multi-armed bandits," *arXiv preprint arXiv:1803.06971*, 2018.

[31] Open Networking Foundation, "Open network operating system (ONOS)," accessed Jul. 29, 2021. [Online]. Available: https://opennetworking.org/onos/

[32] X. Sun, T. E. Ng, and G. Wang, "Software-defined flow table pipeline," in *Proc. IEEE IC2E*, Mar.

[33] T. Benson, A. Akella, and D. A. Maltz, "Data set for IMC 2010 datacenter measurement," accessed Jul. 29, 2021. [Online]. Available: http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html

[34] ——, "Network traffic characteristics of datacenters in the wild," in *Proc. ACM SIGCOMM*, Nov 2010.

[35] Arris enterprises LLC., "Brocade FastIron monitoring configuration guide," accessed Jul. 29, 2021. [Online]. Available: http://docs.ruckuswireless.com/fastiron/08.0.60/fastiron-08060-monitoringguide/GUID-F46B8130-C143-4BDD-AE04-E64821A1A288.html

[36] A. Roy, H. Zeng, J. Bagga *et al.*, "Inside the social network's (datacenter) network," in *Proc. ACM SIGCOMM*, Aug 2015.

[37] S. Knight, H. Nguyen, N. Falkner *et al.*, "The internet topology Zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, 2011.

[38] Y. Li, R. Miao, H. H. Liu *et al.*, "HPCC: High precision congestion control," in *Proc. ACM SIGCOMM*, Aug 2019.

[39] Gurobi Solver, accessed Jul. 29, 2021. [Online]. Available: https://www.gurobi.com

[40] J. S. Rojas, Á. R. Gallón, and J. C. Corrales, "Personalized service degradation policies on OTT applications based on the consumption behavior of users," in *Proc. ICCSA*, Jul 2018.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort *et al.*, "scikit-learn machine learning in python," accessed Jul. 29, 2021. [Online]. Available: https://scikit-learn.org/stable/

[42] Datatracker, "Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information," accessed Jul. 29, 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7011

[43] H. Xu, S. Chen, Q. Ma *et al.*, "Lightweight flow distribution for collaborative traffic measurement in software-defined networks," in *Proc. IEEE INFOCOM*, Sep 2019.

[44] S. Yoon, T. Ha, S. Kim *et al.*, "Scalable traffic sampling using centrality measure on software-defined networks," *IEEE Communications Magazine*, vol. 55, no. 7, 2017.

[45] Z. Liu, R. Ben-Basat, G. Einziger *et al.*, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proc. ACM SIGCOMM*, Aug 2019.

[46] R. Jang, D. Min, S. Moon *et al.*, "Sketchflow: Per-flow systematic sampling using sketch saturation event," in *Proc. IEEE INFOCOM*, Jul 2020.

[47] F. Raspall, "Efficient packet sampling for accurate traffic measurements," *Elsevier Computer Networks*, vol. 56, no. 6, 2012.

[48] D. R. Teixeira, J. M. C. Silva, and S. R. Lima, "Deploying time-based sampling techniques in software-defined networking," in *Proc. IEEE SoftCOM*. IEEE, Sep 2018.

[49] X. Wang, X. Li, S. Pack *et al.*, "STCS: Spatial-temporal collaborative sampling in flow-aware software-defined networks," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 6, 2020.

[50] J. Zhang, X. Luo, R. Perdisci *et al.*, "Boosting the scalability of botnet detection using adaptive traffic sampling," in *Proc. ACM ASIA-CCS*, Mar 2011.

[51] T. Ha, S. Kim, N. An *et al.*, "Suspicious traffic sampling for intrusion detection in software-defined networks," *Elsevier Computer Networks*, vol. 109, no. 2, 2016.

[52] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio *et al.*, "Analysis of the impact of sampling on NetFlow traffic classification," *Elsevier Computer Networks*, vol. 55, no. 5, 2011.

[53] S. Kim, S. Yoon, and H. Lim, "Deep reinforcement learning-based traffic sampling for multiple traffic analyzers on software-defined networks," *IEEE Access*, vol. 9, no. 1, 2021.