Viper: Prioirity-based High-Visibility Per-flow Packet Sampling for SDNs

***Abstract*—Network monitoring is crucial for managing datacenter networks, serving fault diagnosis, traffic measurement, and intrusion detection functions. However, traditional sampling techniques, such as those based on sketches or ports, either lack packet-level granularity or provide insufficient visibility, leading to functional performance degradation. Recent research has employed the software-defined networking (SDN) model to enable flow-based packet sampling. However, these approaches often introduce substantial control and computation overhead, limiting their scalability. This paper presents Viper, a novel priority-based, high-visibility per-flow packet sampling mechanism tailored to address these challenges. Specifically, Viper leverages existing priority-based traffic scheduling mechanisms to prioritize shorter flows over longer ones. Then, a logical central controller refines sampling policies for packets of different priorities. In-depth analysis indicates that the coordinated optimization performed by the controller significantly impacts Viper's performance. Consequently, we model this control process as a nonlinear optimization problem, seeking to maximize the utility of sampling. Then, we propose an online primal-dual interior-point algorithm to address this optimization problem and prove the algorithm's convergence, optimality, and efficiency. Experimental results show that Viper increases visibility by 3.83% to 8.3%, with negligible control overhead and a substantial reduction in sampling load of at least 20.51%.**

***Index Terms*—Packet Sampling, Visibility, Prioirity, Software Defined Networking**

I. Introduction

In today's internet ecosystem, datacenter networks have emerged as a cornerstone of infrastructure. Ensuring their reliability, security, and high performance is paramount. Network monitoring plays a pivotal role in this effort, actively observing and analyzing network traffic packet transmission to promptly identify and resolve issues. However, as modern datacenter networks expand in scale, complexity, and traffic volume, traditional all traffic monitoring approaches, like deep packet inspection (DPI), face scalability challenges and incur significant costs [1]. Packet sampling, a technique that examines a representative subset of network packets rather than involving every single packet that traverses the network, offers a cost-effective solution to this challenge. Therefore, it has becomes an essential conpomnent for many datacenter network management tasks, such as billing [2], traffic scheudling [3], [4], fault diagnosis [5], traffic measurement and flow statistic inference [6], [7], anomaly and intrusion detection [8], [9] and traffic classification [10]. Therefore, an efficient packet sampling framework is essential for modern datacenter networks.

We classify existing packet sampling frameworks into three categories:

- **Per-port packet sampling**, e.g., sFlow [11], cSamp [12], and OpenSample [1], involves sampling packets according to a predetermined rate or policy as they traverse the switch's in-port. Although per-port packet sampling frameworks, especially sFlow [11], are widely supported by legacy switches. They suffer from low visibility problems that short flows, consisting of only a few packets, may be entirely omitted from the sample, as longer flows with higher packet rates and traffic volumes are more likely to be sampled [13]. This limitation makes these methods unsuitable for various network management tasks, especially intrusion detection [14].
- **Per-flow statistics sampling**, e.g., Netflow [15], Elasticsketch [16], HeteroSketch [17], SketchVisor [18], Nitrosketch [19] and SketchFlow [20], maintain a flow cache that tracks per-flow statistics by measuring or even inferring from sampled packets. These methods are mainly designed for functions requiring flow-level information, e.g., network traffic matrix measurement and inference. Although these methods, especially Netflow, are supported by many legacy switches, they cannot fetch packet-level information. This limitation also makes these methods unsuitable for various network management tasks requiring packet-level information for in-depth analysis, such as malware traffic detection [21] and traffic classification [22].
- **Per-flow packet sampling**, e.g., [23], [24] and FlowShark [13], either leverage additional ASIIC logic or Software-Defined Networking (SDN) paradigm with extension OpenFlow protocols to achieve per-flow packet forwarding and sampling. Per-flow packet sampling further solves this problem. These methods, especially Sample-on-Demand [24] and FlowShark [13], leverage the SDN paradigm, which natively supports per-flow packet forwarding operations and facilitates per-flow packet sampling. Although the current OpenFlow does not support packet sampling, many proposals [25], [26] have added per-flow packet sampling extensions to it. However, these methods suffer from scalability issues because they introduce tremendous control and computational overhead for calculating per-flow sampling policies.

This paper presents Viper, a novel priority-based, high-visibility sampling mechanism tailored for flow-specific packet sampling in Software-Defined Networks (SDNs). Viper's key idea is to resort to existing priority-based network traffic scheduling methods to prioritize short flows over long flows, and leverage a logical central controller optimizes per-flow sampling policies for packets with specific priorities, on data and control planes respectively. Specifically, Viper operates by employing mechanisms implemented across three distinct network elements: End hosts apply a demoting priority tagging

mechanism, which tags packets with demoting priorities according to their sent bytes, thus differentiating between short and long flows. SDN switches utilize a priority-based probabilistic sampling mechanism, sampling packets with predetermined probabilities related to their priorities. The controller coordinates these mechanisms through a collaborative control mechanism to facilitate routing decisions and related parameters, including demoting thresholds and sampling probabilities. In-depth analysis indicates that the collaborative optimization performed by the controller significantly impacts Viper's performance. Consequently, we model this control process as a nonlinear optimization problem, seeking to maximize sampling utility. Then, we propose an online primal-dual interior point method to address this optimization problem and provide mathematical proof of the algorithm's convergence, optimality, and computational efficiency. The main contributions of this paper are listed as follows:

- We propose a priority-based, high-visibility per-flow packet sampling framework for SDNs, called Viper. It avoids the control overhead for packet rate estimation and long flow detection by resorting to existing priority-based traffic scheduling methods on the data plane.
- We model Viper's collaboratively control process as a nonlinear optimization problem aiming to maximize sampling utility and propose an online primal-dual interior point method to address it.
- We further provide mathematical proof of the proposed algorithm's convergence, optimality, and computational complexity.
- We conduct testbed and simulation experiments against state-of-the- art methods. Experimental results show that Viper increases visibility by 3.83% to 8.3%, with negligible control overhead and a substantial reduction in sampling load of at least 20.51%.

The organization of this paper is listed as follows: Section II presents the paper's motivation. Section III elaborates on the design of Viper. Section IV formulates the optimal collaborative control problem as a constrained nonlinear optimization problem and proposes a primal-dual interior-point algorithm to solve this problem with proven optimality and convergence. Section V evaluates Viper against the state-of-the-art methods with both small-scale testbed and large-scale simulation experiments. Related work is discussed in Section VI. Finally, Section VII concludes the paper.

II. Motivation

This section presents the motivation and how we are inspired by existing work. First, we illustrate the low visibility issue of short flows in per-port packet sampling frameworks. Then, we present the high control overhead of existing per-flow packet sampling frameworks. Finally, we explain how existing priority-based traffic scheduling methods inspire us.

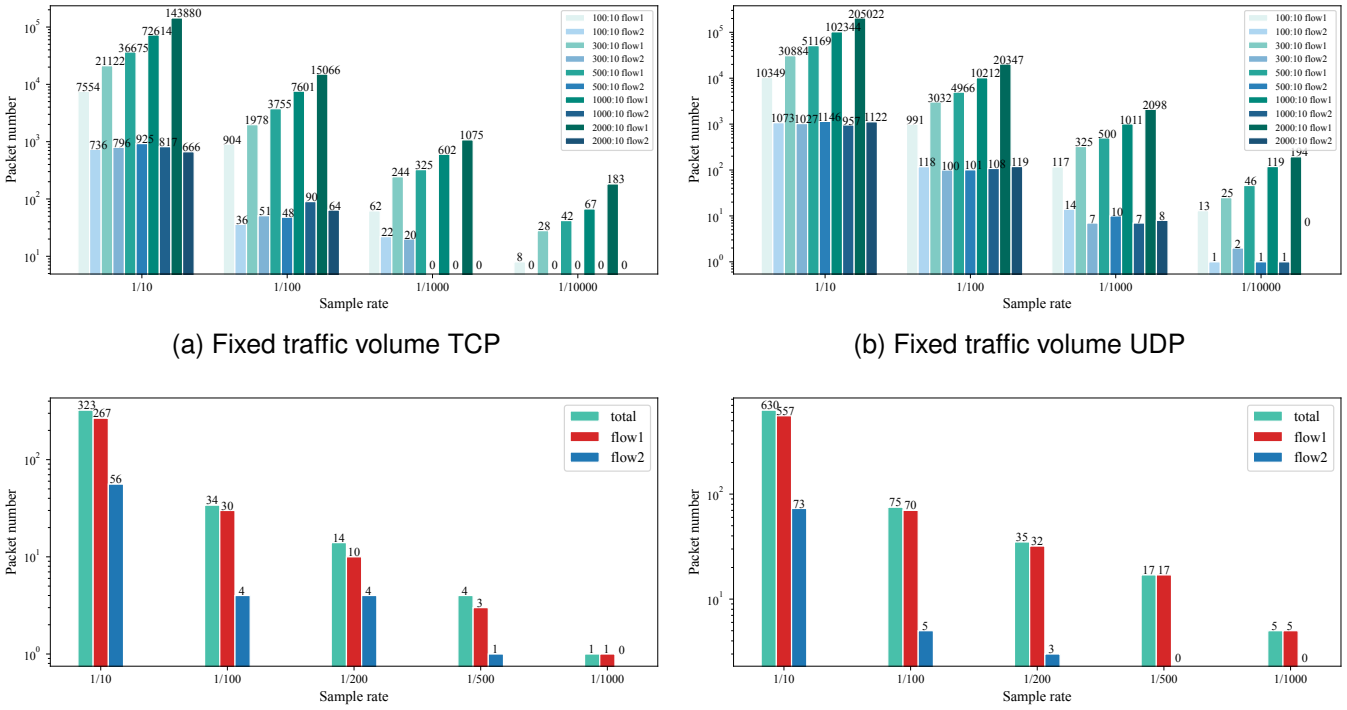
A. Short flows are Less Visibile

We validate the invisibility issue of short flows through a small-scale experiment. Specifically, we utilize Mininet to simulate a network topology featuring two end hosts connected by an SDN switch. We deploy on sFlow on that switch to sample packets and send TCP and UDP packets between end hosts in two distinct modes: fixed packet transmission rate and fixed traffic volume modes. We create a long and short flow between the end hosts in the former mode. To highlight the invisibility of short flows, we fix the traffic volume of the short flow as 10Mb, and adjust the traffic volume of the long flow from 100 MB to 2000 MB, while counting the sampled packet number for both flows. Figures 1a and 1b present the sampled packet number for both flows under sample rates ranging from 1/10 to 1/10000. We find that the sampled packet number for both flows decreases with the decreasing sample rate. Under the same sampling rate, the sampled packet number for the long flow increases with with the increasing traffic volume. In stark contrast, the sampled packet number for the short flow remains when the sample rate is high (1/10 and 1/100) but vanishes when the sample rate is relatively low (1/1000 and 1/10000). In the latter mode, we also generate a long flow and short flow between end hosts, setting packet transmission rates as 100 and 10 packets per second, respectively. Fig. 1c and Fig. 1d show the sampled packet number under sample rate varying from 1/10 to 1/1000, which exhibits a trend analogous to that in the first mode. In summary, the above experiments validate that short flows are less visible.

B. High Control Overhead in Packet Rate Estimation

Since OpenFlow switches inherently support per-flow management and packet forwarding in the SDN paradigm, studies such as [25] and [26] have proposed OpenFlow extensions for per-flow packet sampling. Nonetheless, these approaches must grapple with the challenge of sampling rate allocation [24]. To tackle this issue, existing studies, e.g., [24] and [13], have proposed frameworks that employ a centralized controller to determine the sampling probability for each flow's packets. Specifically, [24] proposes a component on the SDN controller that allows the controller to determine the sampling rate for each flow at each switch according to its instantaneous packet rate. However, such a per-flow sampling rate allocation process results in substantial control and computational overhead. FlowShark [13] mitigates computational overhead by utilizing static wildcard sampling policies with relatively higher sampling rates for short flows' packets but dynamic per-flow sampling rate for long flows' packets. Further investigations reveal that these methods depend on additional modules to gauge per-flow information, which the controller uses to craft optimal sampling probabilities. For instance, [24] suggests using OpenTM [27], while [13] employs a packet estimator in conjunction with a long flow detector, employing machine learning techniques to estimate real-time packet rates and detect long flows.

However, packet rate estimators suffer from accuracy issues and high computational and control overhead. As statistics indicate in [28], the average flow completion time in modern datacenter networks can be in the microseconds range. According to [27], OpenTM requires almost 50 seconds for the estimated packet rates to stabilize, a duration that far exceeds the flow completion time. It inevitably leads to an unneglectable



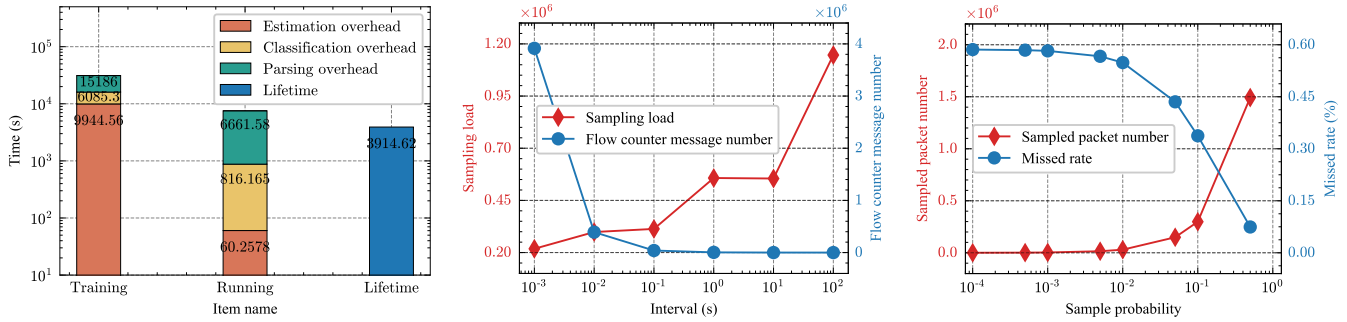
(a) Fixed traffic volume TCP

(b) Fixed traffic volume UDP

(c) Fixed packet transmission rate TCP

(d) Fixed packet transmission rate UDP

Fig. 1. Simulation results for the network.



(a) Training and running overhead

(b) Load and control overhead.

(c) Load and miss ratio.

Fig. 2. The computational and control overhead of existing FlowShark. (a) shows the high running and training computational overhead of FlowShark's packet rate estimator and long flow detector. (b) shows the trade-off between sampling load and control overhead for periodic polling flow counters. Long flow counter polling interval decreases control overhead but leads to excessive sampling load, which causes switch performance degradation. (c) shows the trade-off between sampling rate and sampling load. Low sampling rates could solve the excessive sampling load problem but risk missing short flows.

deviation between the estimated and realistic packet rates, which further causes sub-optimal sampling rate allocation. Then, we evaluated the computational and control overhead of FlowShark by implementing its packet rate estimator and long flow detector using the scikit Python library. We tested them on a server equipped with a 32-core I9-13900 CPU and 128 GB memory using datacenter packet traces from [29]. As shown in Fig. 2a, the running and training time is 1.92 and 7.97 times the lifetime of the whole trace, respectively.

Although more computing resources could mitigate these prolonged training and running times, the control overhead for polling flow counters remains significant. If a switch dispatches a flow counter message upon receiving a TCP/UDP packet, the switch will generate 14,720,318 messages in 3914.62 seconds. Periodic polling flow counters efficiently reduce such control overheads. As Fig. 2b shows, increasing the polling interval to 100 ms drastically cuts the control message number to only 39,155. However, Flowshark might incorrectly treat long flows as short during such an interval. High rate sampling of long flows leads to excessive sampling loads, which causes switch performance degradation [1]. Reducing the sampling rates of static wildcard sampling policies could solve that problem but risks missing short flows. Fig. 2c depicts the packet number sampled by static wildcard sampling policies and flow miss ratio under 100

ms interval[1]. The sampled packet number plummets with the decrease in sampling rate, while the flow miss ratio shows an opposite trend. Although [13] proposes to deploy the long flow detector and rate estimator on edge switch for efficiency through switch modification or programmable switches, such an implementation introduces new compatibility, portability, and scalability issues.

C. Priority-based Scheduling and ECN-enabled Rate Control

Moreover, priority-based traffic scheduling frameworks, combined with Explicit Congestion Norification (ECN) enabled protocols, are widely deployed in modern datacenter networks to enhance performance in terms of end-to-end latency, flow completion time, and deadline guarantee ratio. PIAS, an information-agnostic traffic scheduling framework, minimizes flow completion time by approximating short-flow-first scheduling orders without prior knowledge of traffic volume information. It employs two key mechanisms: the demoting priority tagging mechanism at end hosts and the Multiple Level Feedback Queue (MLFQ) mechanism at switches. The demoting priority tagging mechanism assigns demoting priorities to flows according to the number of bytes it sends. Then, the MLFQ mechanism, implemented through multiple priority queues in existing commodity switches, classifies packets with different priority tags into separate queues and transmits only the head packet of the highest non-empty priority queue. Simultaneously, it mitigates the impact of parameter mismatch by utilizing ECN-enabled protocols for fast congestion and stable rate control.

DCTCP, a representative ECN-based congestion control mechanism, ensures low latency for short flows and high throughput for long flows. In DCTCP, the sender sets the ECN Capable Transport (ECT) codepoint in the IP header to indicate that the ECN mechanism is enabled. ECN-supported switches monitor their buffer space and set the Congestion Encountered (CE) codepoint in the IP header of the outgoing packet if the queue occupancy exceeds a certain threshold after receiving it. Upon receiving a CE codepoint labeled packet, the receiver sets the ECN-Echo flag in a series of ACK packets to the sender until it receives confirmation from the sender via the Congestion Window Reduced (CWR) flag that it has received the congestion notification. Upon receiving ECN-Echo flag marked ACK packets, the sender reduces its congestion window size and responds to the receiver with CWR flag marked ACK packets, thereby avoiding potential congestion. ECN is a compatible, scalable, lightweight, and efficient congestion control protocol. Additionally, by indicating potential congestion before it occurs, ECN allows the source to proactively reduce its transmission rate, avoiding packet loss and sudden throughput decreases. Consequently, ECN promotes a reliable data transfer process with a consistent and stable transmission rate.

Therefore, we propose a packet sampling framework that leverages the SDN paradigm to achieve per-flow packet sampling, the demoting priority tagging mechanism to approximately classify short and long flows without prior knowledge of traffic volume information, and the ECN-enabled protocol and the MLFQ mechanism to control transmission rate and schedule packets, facilitating the estimation of sampling loads on switch queues. This system can accomplish high-visibility per-flow packet sampling without the control overhead mentioned above by developing sampling strategies tailored to each flow's priorities.

[1]In this experiment, we assume that flows with sampling policies determined by the controller are all visible.

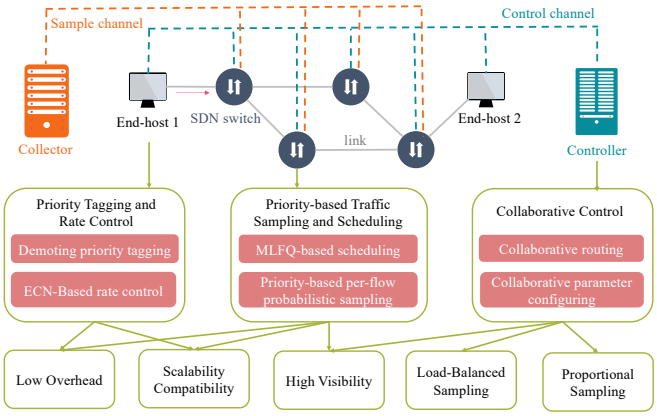


Fig. 3. System architecture, mechanisms and objectives.

III. The Viper Design

In this section, we delve into Viper's design. First, we overview Viper's architecture and components. Second, we detail the design of the priority tagging mechanism at end hosts, the priority-based sampling and traffic scheduling mechanism at network switches, and the cooperative control mechanism at the controller. Finally, we explore Viper's advantages and the criticism of optimal collaborative control.

A. System Overview

Before discussing the overall system architecture and the components of Viper, we outline the design goals of Viper:

- High-Visibility: Viper ensures per-flow level sampling while ensuring each flow is visible within the sample set.
- Scalability and Compatibility: Viper is designed to scale seamlessly within large-scale datacenter networks while being compatible with current commodity switches and protocol stacks.
- Low Overhead: Viper aims to minimize computational overhead, particularly in estimating per-flow packet transmission rate and sent traffic volume.
- Load-Balanced Sampling: Viper can distribute the sampling load evenly across all switches to balance the load.
- Proportional Sampling: Viper ensures that each flow's sampled traffic is proportional to its total traffic volume.

The architecture of Viper is illustrated in Figure 3. It employs an SDN architecture that separates the control and data plane for efficient per-flow network management. When a new flow is detected, the ingress switch encapsulates its first packet and forwards it to the controller for parsing. The controller then determines its routing path and sample policies

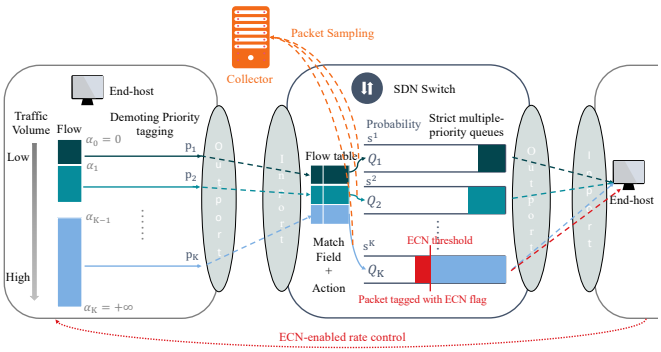


Fig. 4. The working procedure of both end hosts and switches.

by installing flow entries on specific switches. As a result, switches along the path will sample and forward the flow's subsequent packets accordingly. Whenever a switch samples a packet, it encapsulates it and forwards it to the sampling server, known as the collector. To achieve the above goals, Viper relies on specialized mechanisms implemented on three types of network elements within our system:

- End hosts perform demoting priority tagging and ECN-enabled rate control mechanisms to distinguish short and long flows and achieve stable per-flow packet rates.
- Switches perform priority-based per-flow probabilistic sampling and strict MLFQ-based scheduling mechanisms to prioritize short flows over long flows while sampling.
- Controller performs a collaborative control mechanism to coordinate the entire network by setting routing and sampling policies and configuring parameters.

In the following subsection, we will discuss the details of the mechanisms we have designed on these network elements.

B. Priority Tagging and Rate Control at end hosts

Demoting Priority Tagging Mechanism. Viper performs distributed packet priority tagging at the protocol stack of each end host according to each flow's sent traffic volume. Without loss of generality, we define K priorities, represented as $q_1, ...q_K$, satifying $q_1 > ... > q_k > ... > q_K$. As Fig. 4 shows, whenever a new flow arrives, the source end host first assigns it the highest priority q_i. As more bytes are transmitted, once the sent traffic volume of this flow exceeds a pre-defined threshold α_k, the subsequent packets of this flow will be tagged with decreasing priorities q_k $(2 < k < K)$ and scheduled accordingly in the network. We refer to α_k as the demotion threshold that seperates two adjacent priorities q_k and q_{k+1} and define $\alpha_0 = 0 < \alpha_1 < ... < \alpha_k < ... < \alpha_K = +\infty$. Meanwhile, end hosts also periodically report the traffic volume information of their completed flows to the controller for statistics.

ECN-based Rate Control Mechanism. Moreover, Viper utilizes ECN-enabled TCP protocols, e.g., DCTCP [28], on end hosts to control the transmission rate of each flow for low latency. The destination end host responds to the exact sequence of marked packets back in two alternative ways: First, the destination end host will send an ACK packet for every packet, setting the ECN-Echo flag if the packet has a marked CE codepoint. Second, the destination end host send will send one ACK packet for conservative packets marked with the same CE codepoint, setting the ECN-ECho flag according to the CE codepoint value of these packets. Whenever the destination end host receives two adjacent packets with different CE codepoints, it immediately sends an ACK packet to the source end host, setting the ECN-ECho flag the same as the CE codepoint of the later packet. The source end host will maintain an estimate of the fraction of packets marked CE codepoint according to the number of ACK packets marked ECN-ECho flag. Then, it cuts the flow's window size proportionally to such fraction value. Consequently, end hosts can adjust their flow transmission rates to avoid potential network congestion, achieving a relatively stable per-flow packet transmission rate.

C. Priority-based Sampling and Scheduling at Switches

MLFQ-based Traffic Scheduling Mechanism. Viper performs distributed priority-based per-flow packet sampling and traffic scheduling at each switch. Specifically, each switch in Viper adopts a strict K-MLFQ to schedule packets according to their priorities tagged on end hosts, denoted as $Q_1, ...Q_k, ...Q_K$. Flow entries installed on each switch are responsible for classifying packets with priority q_k into queue Q_k. The transmission of the head packet from Q_k can only proceed unless queues with higher priority are empty.

Priority-based Per-flow Probabilistic Sampling Mechanism. In addition to this structured queuing traffic scheduling mechanism, Viper also incorporates a priority-based per-flow probabilistic sampling mechanism on switches by associating flow entries that classify packets into Q_k with a sampling probability s_k, indicating the likelihood that a packet traversing through Q_k will be selected for sampling. For a sampled packet, the switch will send a copy of it to the remote packet sampling server. Consequently, a switch will sample each flow's packets belonging to $(\alpha_{k-1}, \alpha_k]$ traffic volume in Q_k with probability s_k. Therefore, Viper can ensure comprehensive visibility in the sample sets by properly configuring α and s on across all switches. Besides, switches also support ECN. They can mark the arriving packet's CE label if the length of the queue this packet belongs is larger than the predefined threshold, enabling end hosts to dynamically manage flow sending rate to prevent network congestion. Fig. 4 shows the working procedure of both end hosts and switches.

D. Collaborative Control at Controller

Collaborative Control Mechanism. In Viper, the network controller maintains an authoritative global overview of the network. It collaboratively controls all end hosts and switches in the network by setting routing and sampling rules, as well as configuring relevant parameters, e.g., demotion thresholds and sampling probability. Intitively, the collaborative control mechanism rely on a routing module and a parameter configuation module working cooperatively. Whenever a new flow accesses the network, the ingress switch encapsulates its first packet as a PACK-IN message, then dispatches it

to the controller requesting to establish routing and sampling rules for that flow. Upon receiving this message, the controller parses it and then invokes the routing module to determine the optimal routing path and the parameter configuation module to calculate the sampling probability for each queue of all switches on the path. Then, the controller installs these routing and sampling rules on switches along the path as flow entries. Finally, switches along the routing path will forward and sample the subsequent flow packets accordingly. In addition, the parameter configuation module is also responsible for collaboratively configuring the demotion thresholds on the end hosts according to the dynamic of traffic volume distribution.

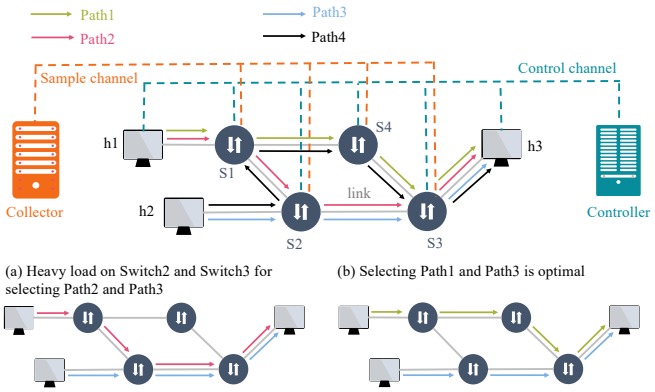


Fig. 5. Illustration example of optimal routing. There are two alternative routing paths for flow1 and flow2, path1 and path2 for flow1, path3 and path4 for flow2. As case (a) shows, if Viper selects path2 for flow1 and path3 for flow2. Switch2 and switch3 will bear heavy sampling loads for sampling both flows. Similar situations occur if Path1 and Path4 (Leaving heavy loads on Switch1, Switch3, and Switch4) and Path2 and Path4 (Leaving heavy loads on Switch1, Switch2, and Switch3) are selected. While case (b) presents the optimal routing.

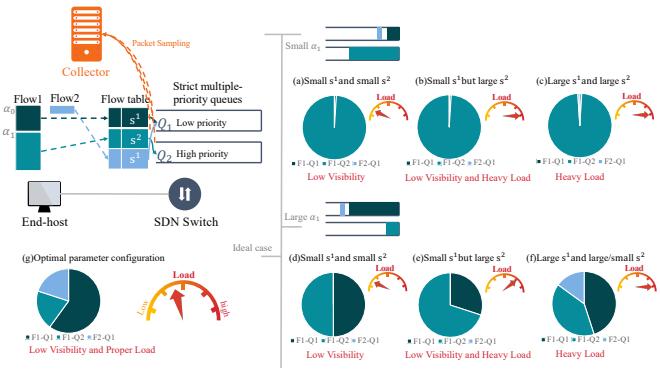


Fig. 6. Illustration example of optimal parameters configuration: (a) Small α_1, s^1 and s^2 cause low visibility problem for flow2; (b) The low visibility and heavy load problem caused by small α_1, s^1 and large s^2; (c) The heavy load problem caused by small α_1 and large s^1 and s^2;(d) Large α_1, small s^1 and s^2 cause low visibility problem for flow2; (e) The low visibility and heavy load problem caused by large α_1 and s^2 but large s^1; (f) The heavy load problem caused by large α_1 and s^1.

E. Discussion

The Advantages of Viper Design. Firstly, the priority demotion tagging mechanism allows flows to dynamically adjust their priorities based on the number of bytes they send. When packets arrive at the network, switches employ an MLFQ-based scheduling mechanism to process them, prioritizing short flows in the initial queues and long flows in lower-priority queues. This approach enables Viper to prioritize short flows over long flows without continuously polling flow counters.Moreover, each switch incorporates a per-flow priority-based probabilistic sampling mechanism that samples packets based on the flow entries corresponding to flows' priorities. This feature ensures that Viper is efficient and adaptable, maintaining compatibility with existing commodity SDN switches and protocol stacks. Lastly, since the controller maintains the global network view and determines every new flow's routing path by parsing its first packet, it can promptly obtain the overall flow arrival rate of the entire network and determine the flow arrival rate of each switch. Consequently, it can estimate the sampling load of each switch, even the sampling load of each queue of each switch, with a roughly estimated overall traffic volume distribution, thus avoiding the complicity of estimating per-flow packet rate. By adjusting sampling probabilities inversely with priority levels, Viper achieves three benefits: almost equal visibility for short and long flows, sampling load balance, and proportional sampling.

The Criticism of Collaborative Control. The Viper controller manages the network by setting routing and sampling policies on switches and configuring demoting thresholds on end hosts. Consequently, the routing and sampling policies, as well as parameter configuration, can significantly impact the performance of the whole system. To illustrate the effects of optimal routing and sampling policies, consider a network scenario where two end hosts (host1 and host2) generate traffic flows (flow1 and flow2) destined for another host (host3). As depicted in Fig. 5, multiple routing options are available: path1 and path2 for flow1, and path3 and path4 for flow2. If Viper chooses path2 for flow1 and path3 for flow2 (case a), switch2 and swithc3 will experience high sampling loads due to both flows. Although Viper can alleviate some of flow1's sampling load from switch2 to switch1 by raising the latter's sampling probability, there is no equivalent solution for the load on switch2 and switch3 from flow2. Similar situations occur if Path1 and Path4 (Leaving heavy loads on Switch1, Switch3, and Switch4) and Path2 and Path4 (Leaving heavy loads on Switch1, Switch2, and Switch3) are selected. In case b, the optimal routing would be selecting path1 for flow1 and path3 for flow2, even though switch3 would still bear sampling loads for both flows. Viper can address this by setting a lower sampling probability on switch3 but higher sampling probabilities on switch1, switch2, and switch4. Clearly, routing and parameter configuration modules work collaboratively to optimize Viper's performance.

We illustrate the criticism of optimal parameter configuration in Fig. 6. Consider a scenario where a source end host generates two flows, flow1 and flow2, towards a destination end host through a switch. Flow1 has a significantly larger traffic volume than flow2. We implement the MPQ mechanism with two queues at the switch egress port. With two priorities, we only need one demotion threshold, α_1, from the high-priority queue to the low-priority queue and two sampling

probabilities, s_1 and s_2, corresponding to the high and low priorities, respectively. In cases (a), (b), and (c), Viper sets a small s_1 causing flow1 to enter the low-priority queue prematurely. In case (a), if Viper also sets small s_1 and s_2, flow2 may be invisible due to its low traffic volume. While simply increasing s^2 cannot solve this problem unless $v2 < \alpha_1$, it merely exacerbates the load on the switch for sampling a large proportion of flow1, as case (b) shows. Case (c) demonstrates that while a large s_1 ensures flow2's visibility, it also leads to the same heavy load problem with a large s_2 as case (b) shows. In case (d), (e), and (f), Viper sets a large α_1, resulting in long flows' packets remaining in the high-priority queue. In case (d), flow2 remains invisible when Viper sets small s_1 and s_2, similar to case (a). However, unlike case (b), increasing s_2 is ineffective in solving this issue but increasing sampling load, as the end host is unlikely to assign flow2 a low priority with a large α_1, as shown in case (e). Case (f) indicates that simply setting a large s_1 ensures flow2's visibility but also reintroduces the heavy load problems observed in cases (b) and (c). The above cases highlight the importance of carefully setting routing paths and sampling policies and adjusting the demoting priority thresholds to improve Viper's efficiency and performance. However, the optimal collaborative control decisions depends on traffic load, e.g., flow arrival rate and flow size distribution, which is highly dynamic, necessitating a dynamic and adaptive algorithm.

TABLE I
NOTATIONS

Symbols	Definitions
$G = (\mathcal{N}, \mathcal{L})$	Network topology directed graph
$\mathcal{N}$	Node set
$\mathcal{L}$	Link set
$\mathcal{P}$	Path set
n_i	The i-th node in $\mathbf{N}$
l_{n_i,n_j}	Link from n_i to n_j
C_{n_i,n_j}	Link capacity of l_{n_i,n_j}
P_i	The i-th path in $\mathbf{P}$
$x_{i,j}$	Indicator of n_i and P_j
λ	Flow arrival rate
K	Number of priorities
$\boldsymbol{p} = (p_1, ..., p_K)$	Priority
$\boldsymbol{\alpha} = (\alpha_0, ..., \alpha_K)$	Demotion threshold
$\boldsymbol{Q} = (Q_1, ..., Q_K)$	K-level MPQ
$s_{i,k}$	Sampling probability of the k-th queue on n_i
r_{n_i,n_j}	Flow arrival rate between n_i and n_j
W_{n_i}	Sampling load constraint on n_i
$F(\cdot)$	Traffic volume CDF
$f(\cdot)$	Traffic volume PDF
R_{n_a,n_b}	Flow arrival rate on link l_{n_a,n_b}
$\rho^k_{n_a,n_b}$	Traffic load of the k-th queue of l_{n_a,n_b}
$m^k_{n_a,n_b}$	Remained bandwidth of the k-th queue on l_{n_a,n_b}
$t^k_{n_a,n_b}$	Transmission rate of the k-th queue on l_{n_a,n_b}
w_{n_a}	Sampling load of n_a
$S^{z,k}_{n_i,n_j}$	sampling probability of the k-th queue on $\mathbf{P}^z_{n_i,n_j}$
P_{n_i,n_j}	Sample ratio of traffic from n_i to n_j
$U(\cdot)$	Utility function

IV. Optimal Collaborative Control

This section first models optimal collaborative control as a constrained nonlinear programming problem. Then, we propose a primal-dual interior-point algorithm to solve this problem. Next, we analyze the performance of our proposed algorithm in terms of optimality, convergence, and complexity. Finally, we discuss some alternative features and algorithm improvements that correspond to the design of Viper.

A. Formulation

We model the network topology as a directed graph $G = (\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ denotes the switch set and $\mathcal{L}$ represents the link set. Each $n_i \in \mathcal{N}$ represents a switch, and each link $l_{n_i,n_j} \in \mathcal{L}$ represents the link from n_i to n_j. Then, we define the link capacity of $l_{n_i n_j}$ as C_{n_i,n_j}, and $|\mathcal{N}|$ and $|\mathcal{L}|$ as the number of total nodes and links, respectively. We further organize all paths between n_i and n_j as a set $\mathcal{P}_{n_i,n_j} = \{\mathbf{P}^1_{n_i,n_j}, ..., \mathbf{P}^{|\mathcal{P}_{n_i,n_j}|}_{n_i,n_j}\}$ where $|\mathcal{P}_{n_i,n_j}|$ is the total path number between n_i to n_j and $\mathbf{P}^z_{n_i,n_j}$ is the z-th patch between n_i and n_j, represented as a vector consisting of a series of contiguous links, $\mathbf{P}^z_{n_i,n_j} = (l_{n_i,n_a}, l_{n_a,n_b}, ..., l_{n_x,n_j})$. We futher define the number of links on $\mathbf{P}^z_{n_i,n_j}$ as its length, represented as $|\mathbf{P}^z_{n_i,n_j}|$. Besides, we also define an indicator function $I(\cdot)$ to indicate whether l_{n_a,n_b} is on $\mathbf{P}^z_{n_i,n_j}$, especially if $l_{n_a,n_b} \in \mathbf{P}^z_{n_i,n_j}$, $I(l_{n_a,n_b}, \mathbf{P}^z_{n_i,n_j}) = 1$, otherwise $I(l_{n_a,n_b}, \mathbf{P}^z_{n_i,n_j}) = 0$. Clearly, $\sum^{\mathcal{L}}_{n_a} I(l_{n_a,n_b}, \mathbf{P}^z_{n_i,n_j}) = |\mathbf{P}^z_{n_i,n_j}|$. We summarize notations in Table I.

Each flow comprises packets with a common flow identifier, consisting of several selected fields from the packet header. A typical flow identifier is a 5-element tuple containing the fields of source IP address, source port, protocol type, destination IP address, and destination port. We assume the traffic volume of each flow, v, follows a pre-known distribution and denote its cumulative probability distribution function (CDF) and probability density function (PDF) as $F(\cdot)$ and $f(\cdot)$, respectively. So, we can use $F(\alpha_k) - F(\alpha_{k-1})$ to denote the percentage of flows with traffic volume in $[\alpha_{k-1}, \alpha_k)$. Since the controller maintains the global overview of the network and responses to every PACKET-IN message, it can observe real-time request arrival rate for any pair of source node n_i and destination node n_j in real-time, represented as r_{n_i,n_j}. Therefore, the overall flow arrival rate is $r = \sum^{\mathcal{N}}_{n_i} \sum^{\mathcal{N}}_{n_j} r_{n_i,n_j}$. Recall that the controller is also responsible for making routing decisions. Therefore, it can distribute λ_{n_i,n_j} flows across $\mathcal{P}_{n_i,n_j}$. We further denote the fraction of flows distributed on its paths as a vector, $\mathbf{q}_{n_i,n_j} = (q^1_{n_i,n_j}, ..., q^{|\mathcal{P}_{n_i,n_j}|}_{n_i,n_j})$. Obviously, $|\mathbf{q}_{n_i,n_j}| = \sum^{|\mathcal{P}_{n_i,n_j}|}_{z} q^z_{n_i,n_j} \leq 1$. Therefore, the flow arrival rate on link l_{n_a,n_b} can be calculated by

$$R_{n_a,n_b} = \sum^{\mathcal{N}}_{n_j} \sum^{\mathcal{N}}_{n_i} \sum^{|\mathbf{P}_{n_i,n_j}|}_{z=1} I(l_{n_a,n_b}, \mathbf{P}^z_{n_i,n_j}) \cdot r_{n_i,n_j} \cdot q^z_{n_i,n_j} \quad (1)$$

Representing the traffic volume of flows as v and the expected traffic volume as $\mathrm{E}(v)$, we denote the traffic load on l_{n_a,n_b} as

$$\rho_{n_a,n_b} = R_{n_a,n_b} \cdot \mathrm{E}(v). \quad (2)$$

Recall that end hosts tag flows with priorities according to their sent traffic volume, $F_{\alpha_{k-1}}$ flows will not contain packets tagged with p_k and scheduled in Q_k, and $1 - F(\alpha_k)$ flows

will each generates $\alpha_k - \alpha_{k-1}$ traffic volume in Q_k, while $F(\alpha_k) - F(\alpha_{k-1})$ flows will each generates an expected $\int_{\alpha_{k-1}}^{\alpha_k} \frac{v \cdot f(v)}{F(\alpha_k) - F(\alpha_{k-1})} dv$ traffic volume in Q_k. Therefore, the traffic load of the k-th queue of l_{n_a,n_b} can be represented as

$$\rho^k_{n_a,n_b} = R_{n_a,n_b} \cdot \mathrm{E}_k(v), \tag{3}$$

where $\mathrm{E}_k(v)$ is the expected per-flow traffic volume sent in the k-th queue

$$\mathrm{E}_k(v) = \int_{\alpha_{k-1}}^{\alpha_k} v f(v)\, dv + (\alpha_k - \alpha_{k-1})\big(1 - F(\alpha_k)\big) \tag{4}$$

Since each switch adopts a K-level strict MPQ mechanism to schedule and sample packets, the transmission of the head packet from Q_k can only proceed unless queues with higher priority are empty. Therefore, the remained bandwidth for the k-th queue on l_{n_a,n_b} can be calculated by

$$m^k_{n_a,n_b} = C_{n_a,n_b} - \sum_{i=0}^{k-1} \rho^i_{n_a,n_b}. \tag{5}$$

Thus, the transmission rate of the k-th queue on l_{n_a,n_b} can be represented as

$$t^k_{n_a,n_b} = \min\{m^k_{n_a,n_b}, \rho^k_{n_a,n_b}\}. \tag{6}$$

Leveraging a logically centralized controller, Viper can avoid network congestion through routing control and admission control. Besides, it also achieves fast rate control in response to network congestion by adopting an ECN-based end-to-end rate control mechanism. Hence, we can view the network as congestion-free, so we have $t^k_{n_a,n_b} = \rho^k_{n_a,n_b}$. Given the sampling probability vector on n_a, represented as $\mathbf{s}_{n_a} = (s^1_{n_a}, ...s^K_{n_a})$, we can calculate the sampling load of n_a by

$$w_{n_a} = \sum_{n_j}^{\mathcal{N}_{n_a}} \sum_{k=1}^{K} t^k_{n_a,n_j} s^k_{n_a}, \tag{7}$$

where $\mathcal{N}_{n_a} = \{n_i | \exists\, l_{n_a,n_i} \in \mathcal{L}\}$.

Since flows on $\mathbf{P}^z_{n_i,n_j}$ can be sampled by all switches along the path. Hence, we can calculate the sampling probability of the k-th queue on path $\mathbf{P}^z_{n_i,n_j}$ by

$$S^{z,k}_{n_i,n_j} = \sum_{l_{n_a,n_b}}^{\mathcal{L}} I(l_{n_a,n_b}, \mathbf{P}^z_{n_i,n_j}) s^k_{n_a}. \tag{8}$$

Then, we define the sample ratio of traffic from n_i to n_j as the proportion of sampled traffic volume to its total traffic volume. It can be calculated by

$$g_{n_i,n_j} = \frac{\sum_{z=1}^{|\mathcal{P}_{n_i,n_j}|} \sum_{k=1}^{K} \mathrm{E}_k(v) q^z_{n_i,n_j} S^{z,k}_{n_i,n_j}}{\mathrm{E}(v)}. \tag{9}$$

Clearly, a higher P_{n_i,n_j} improves the accuracy of network performance and flow statistic measurement, finer granularity of anomaly and intrusion detection, and faster speed of fault diagnosis. Therefore, we define a non-negative monotonically increasing convex function $U_{n_i,n_j}(\cdot)$ to denote the utility obtained for achieving a specific sample ratio for flows between n_i and n_j. Then, we model the collaborative configuration problem of Viper as the following optimization problem, labeled as **P1**

$$\max_{\mathbf{q},\mathbf{s},\boldsymbol{\alpha}} \sum_{n_i}^{\mathcal{N}} \sum_{n_j}^{\mathcal{N}} U_{n_i,n_j}(g_{n_i,n_j}), \tag{10a}$$

$$s.t. \quad 0 \le s^k_{n_i} \le 1,\ \forall n_i \in \mathcal{N}, k \in \{1, ..., K\} \tag{10b}$$

$$0 \le q^z_{n_i,n_j},\ \forall n_i, n_j \in \mathcal{N}, z \in \{1, ..., |\mathcal{P}_{n_i,n_j}|\} \tag{10c}$$

$$|\mathbf{q}_{n_i,n_j}| \le 1,\ \forall n_i, n_j \in \mathcal{N}, \tag{10d}$$

$$\rho_{n_i,n_j} \le C_{n_i,n_j},\ \forall l_{n_i,n_j} \in \mathcal{L}, \tag{10e}$$

$$w_{n_i} \le W_{n_i},\ \forall n_i \in \mathcal{N}, \tag{10f}$$

$$\alpha_{k-1} < \alpha_k,\ \forall k \in \{1, 2, ..., K-1\}, \tag{10g}$$

Here, constraints (10b), (10c), and (10d) ensure the non-negativity and normalization of probability, constraints (10e) ensure the traffic load on each link is lower than its capacity, constraints (10f) ensure the sampling load on n_i is lower than its sampling load constraints B_{n_i}, while constraints (10g) ensure Viper assgins flows with descent priorities with the increasing of sent traffic volume.

Clearly, **P1** is a constrainted nonlinear optimization problem with $|d| = |\mathcal{N}|^2 + (2|\mathcal{N}| + 1)K + |\mathcal{L}| + \sum_{n_i} \sum_{n_j} |\mathcal{P}_{n_i,n_j}|$ constraints and $|\mathbf{x}| = (|\mathcal{N}|+1)K + \sum_{n_i} \sum_{n_j} |\mathcal{P}_{n_i,n_j}|$ decision variables. It is worth noting that Viper's performance is highly dependent on the optimality of its parameter configuration. Consequently, it is essential to develop an algorithm that can efficiently handle large-scale problems while maintaining global convergence. The primal-dual interior-point method has advantages, including global convergence, efficiency, and parallelizability, making it a preferred choice for our problem.

B. Algorithm

Problem Relaxation. First, we organize all decision variables, including $\boldsymbol{\alpha}$, $\mathbf{q}$ and $\mathbf{s}$, as a vector $\mathbf{x} = (x^1, ...x^{|\mathbf{x}|})$ and rewrite **P1** into standard form, represented as $d_i(\mathbf{x}) \ge 0$. Then, **P1** is equivalent to the following problem, labeled as **P2**

$$\min_{\mathbf{x} \in \mathcal{R}^{|\mathbf{x}|}} U(\mathbf{x}), \tag{11a}$$

$$s.t. \quad d_j(\mathbf{x}) \ge 0, j \in \{1, ..., |d|\}, \tag{11b}$$

where $U(\mathbf{x}) = -\sum_{n_i}^{\mathcal{N}} \sum_{n_j}^{\mathcal{N}} U_{n_i,n_j}(g_{n_i,n_j})$. According to the Lagrange duality, we can further relax **P2** by constructing a Lagrangian relaxed problem, labeled as **P3**

$$\max_{\boldsymbol{\lambda}} \min_{\mathbf{x} \in \mathcal{R}^{|\mathbf{x}|}} L(\mathbf{x}, \boldsymbol{\lambda}) = U(\mathbf{x}) - \sum_{j=1}^{|d|} \lambda_j d_j(\mathbf{x}), \tag{12a}$$

$$s.t. \quad \lambda^j \ge 0, j \in \{1, ..., |d|\}, \tag{12b}$$

where $L(\mathbf{x}, \boldsymbol{\lambda})$ is referred to as Lagrangian and $\boldsymbol{\lambda} = (\lambda^1, ..., \lambda^{|d|})$ is the vector of Lagrange multipliers. Obviously, we can find stationary points of **P3** by solving $\frac{\partial L(\mathbf{x},\boldsymbol{\lambda})}{\partial \mathbf{x}} = 0$ and $\frac{\partial L(\mathbf{x},\boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}} = 0$, which is extremely complicated for solving a high-dimensional non-linear equation system.

Proposition 1: **P3** is the strong dual problem of **P2**.

Proof: The proof can be referred to in the Appendix. ■

Then, according to [30], we transform **P2** with an equivalent logarithmic barrier function

$$B(\mathbf{x}, \boldsymbol{\mu}) = U(\boldsymbol{x}) - \sum_{i=1} \mu_i \log d_i(\mathbf{x}). \tag{13}$$

Here, $\boldsymbol{\mu} = (\mu_1, ..., \mu_{|d|})$ where $\forall \mu_j > 0$. Clearly, **P2** can be solved by iteratively solving a sequence of unconstrained $B(\mathbf{x}, \boldsymbol{\mu})$ minimization problems until $\boldsymbol{\mu} \to \mathbf{0}$. We further derive the gradient of $B(\boldsymbol{x}, \boldsymbol{\mu})$ as

$$\nabla_{\mathbf{x}} B(\mathbf{x}, \boldsymbol{\mu}) = u(\mathbf{x}) - J_d(\mathbf{x})^T D_d(\mathbf{x})^{-1} \boldsymbol{\mu} \tag{14}$$

Here, $u(\mathbf{x})$ is the gradient of $U(\mathbf{x})$, while $J_d(\mathbf{x})$ and $D_d(\mathbf{x})$ are the Jacobian and diagonal matrix of $d_i(\mathbf{x})$, respectively. Note that $\nabla_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}) = u(\mathbf{x}) - J_d(\mathbf{x})\boldsymbol{\lambda}$. Hence we can approximate $\nabla_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda})$ by approximating $\boldsymbol{\lambda}$ with $D_d(\mathbf{x})^{-1}\boldsymbol{\mu}$, represented as $\boldsymbol{\lambda} = D_d(\mathbf{x})^{-1}\boldsymbol{\mu}$. Therefore, according to the Kuhn-Tucker (KT) first order necessary condiction of optimality, we can find KT points of $L(\mathbf{x}, \boldsymbol{\lambda})$ by solving the following equation system

$$\begin{cases} u(\mathbf{x}) - J_d(\mathbf{x})^T \boldsymbol{\lambda} = 0, \\ D_d(\mathbf{x})\boldsymbol{\lambda} = \boldsymbol{\mu}. \end{cases} \tag{15}$$

Newton Iteration. Adopting a Newton iteration method for the solution of (15), we have the following Newton equations

$$\Omega(\mathbf{x}, \boldsymbol{\lambda}) \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} u(\mathbf{x}) - J_d(\mathbf{x})^T \boldsymbol{\lambda} \\ \boldsymbol{\mu} - D_d(\mathbf{x})\boldsymbol{\lambda} \end{bmatrix} \tag{16}$$

where

$$\Omega(\mathbf{x}, \boldsymbol{\lambda}) = \begin{bmatrix} -H(\mathbf{x}) & J_d(\mathbf{x})^T \\ D_\lambda(\boldsymbol{\lambda}) J_d(\mathbf{x}) & D_d(\mathbf{x}) \end{bmatrix} \tag{17}$$

Here, $H(\mathbf{x})$ is the Hessian of $L(\mathbf{x}, \boldsymbol{\lambda})$ and $D_\lambda(\boldsymbol{\lambda})$ is the diagonal matrix consisting of λ_i. First, we randomly select a feasible solution x_0 from the feasible solution set $\mathcal{X} = \Big\{\mathbf{x} \Big| \forall i \in \{1, ..., |d|\}, d_i(\mathbf{x}) \geq 0\Big\}$. Then, we iterate $\mathbf{x}$ and $\boldsymbol{\lambda}$ for I iterations or until $|\Delta\mathbf{x}_i|$ is lower than a nonnegative value ϵ. For the i-th iteration, given $\mathbf{x}_i$ and $\boldsymbol{\lambda}_i$, we can obtain $\Delta\bar{\mathbf{x}}_i$ and $\Delta\bar{\boldsymbol{\lambda}}_i$ by solving (16) with $\bar{\boldsymbol{\mu}}_i = \mathbf{0}$. Obviously, $\Delta\bar{\mathbf{x}}_i$ is a descent direction for **P2** at $\mathbf{x}_i$.

***Proposition** 2:* Given any vecor $\mathbf{x}_i \in \mathcal{X}$, any positive definte matrix $H(\mathbf{x}_i)$, and any nonnegative vector $\boldsymbol{\lambda}_i$ such that $\lambda_i^j > 0$ if $d_j(\mathbf{x}_i) = 0$, the matrix $\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i)$ is nonsingular.

Proof: The proof can be referred to in the Appendix. ■

Descent Direction Correction. However, $\Delta\bar{\mathbf{x}}_i$ is not an entirely suitable search direction, because if $\exists\, d_j(\mathbf{x}) \to 0$, according to (16), we have $\lambda_i^j \nabla_{\mathbf{x}} d_j(\mathbf{x}) \Delta\bar{\mathbf{x}}_i + d_j(\Delta\bar{\mathbf{x}}_i)(\lambda_i^j + \Delta\lambda_i^j) = 0$. Therefore, $\Delta\bar{\mathbf{x}}_i$ tend to be a direction tangent of $\mathcal{X}_f$, which may cause feasibility violation for any step length in $\Delta\bar{\mathbf{x}}_i$. According to [30], we can eliviate this problem by substituting $\bar{\boldsymbol{\mu}}_i = \mathbf{0}$ with

$$\tilde{\boldsymbol{\mu}}_i = ||\Delta\bar{\mathbf{x}}_i||^\sigma \boldsymbol{\lambda}_i. \tag{18}$$

where σ is a constant parameter in $(2, 3)$, and obtaining a new correction direction $\Delta\tilde{\mathbf{x}}_i$ and corresponding $\Delta\tilde{\boldsymbol{\lambda}}_i$ by solving (16) with $\tilde{\boldsymbol{\mu}}_i$. Then, correct $\Delta\mathbf{x}_i$ and $\Delta\boldsymbol{\lambda}_i$ by

$$\Delta\mathbf{x}_i = (1 - \varphi_i)\Delta\bar{\mathbf{x}}_i + \varphi_i \Delta\tilde{\mathbf{x}}_i. \tag{19a}$$

$$\Delta\boldsymbol{\lambda}_i = (1 - \varphi_i)\Delta\bar{\boldsymbol{\lambda}}_i + \varphi_i \Delta\tilde{\boldsymbol{\lambda}}_i, \tag{19b}$$

where

$$\varphi_i = \begin{cases} 1, & \text{if } u(\mathbf{x}_i)\Delta\tilde{\mathbf{x}}_i \leq \theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i, \\ \frac{(1-\theta)u(\mathbf{x})\Delta\bar{\mathbf{x}}_i}{u(\mathbf{x})(\Delta\bar{\mathbf{x}}_i - \Delta\tilde{\mathbf{x}}_i)}, & \text{otherwise.} \end{cases} \tag{20}$$

Here, θ is a pre-defined constant value in $(0, 1)$. Then, the following propositions prove (20) ensures $\Delta\mathbf{x}_i$ is a descent direction.

***Proposition** 3:* $\Delta\mathbf{x}_i$ satifying

$$u(\mathbf{x}_i)\Delta\mathbf{x}_i \leq \theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i \leq -\theta \Delta\bar{\mathbf{x}}_i H(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i. \tag{21}$$

Proof: The proof can be referred to in the Appendix. ■

***Proposition** 4:* If **Proposition 2** holds, then

$$||\Delta\mathbf{x}_i - \Delta\bar{\mathbf{x}}_i|| \leq \lambda_{max} \text{sup}\{||\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i)^{-1}||\}||\Delta\bar{\mathbf{x}}_i||^\sigma \tag{22}$$

where λ_{max} represents the upper bound of λ_i^j.

Proof: The proof can be referred to in the Appendix. ■

Second Order Decsent Direction Correction. However, $\boldsymbol{\mu}$ may become very small when $\mathbf{x}_i$ in the vicinities of stationary points, which will further cause low convergence speed for $\lambda_i^j + \Delta\lambda_i^j \geq d_j(\mathbf{x})$, or even fail in converging to the true Lagrange multiplier vector. Therefore, we define the active constraint set as $\mathcal{A}(\mathbf{x}) = \{i | \lambda_i^j + \Delta\lambda_i^j \geq d_j(\mathbf{x})\}$ and the violated constraint set as $\mathcal{V}(\mathbf{x}) = \{i | \lambda_i^j + \Delta\lambda_i^j \leq -d_j(\mathbf{x})\}$, and involve a Maratos second order correction $\Delta\dot{\mathbf{x}}_i$, which is computed by solving the following linear least squares problem, labeled as **P4**, to achieve a full Newton step to be taken near the stationary points [31].

$$\min\ -\frac{1}{2}\Delta\dot{\mathbf{x}}_i H(\mathbf{x}_i)\Delta\dot{\mathbf{x}}_i, \tag{23a}$$

$$s.t. \quad d_j(\mathbf{x}_i + \Delta\mathbf{x}_i) + \nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\dot{\mathbf{x}}_i = \xi_i, \forall j \in \mathcal{A}(\mathbf{x}_i) \tag{23b}$$

where

$$\xi_i = \max\Big\{||\Delta\mathbf{x}_i||^\sigma, \max_{j \in \mathcal{A}(\mathbf{x}_i)} \Big|\frac{\Delta\lambda_i^j}{\lambda_i^j + \Delta\lambda_i^j}\Big| ||\Delta\mathbf{x}_i||^2\Big\}. \tag{24}$$

Constraints (23b) ensure $d_j(\mathbf{x}_i + \Delta\mathbf{x}_i + \Delta\dot{\mathbf{x}}_i) \geq ||\Delta\mathbf{x}_i||^\theta$, which leads to a superlinear convergence rate for $\boldsymbol{\lambda}$. Besides, $\mathcal{V}(\mathbf{x}) = \emptyset$ when $\mathbf{x}_i$ approaches stationary points. Therefore, we argue to perform such second order decsent direction correction only when $\mathcal{V}(\mathbf{x}) = \emptyset$.

***Proposition** 5:* For i large enough, the second order correction $\Delta\dot{\mathbf{x}}_i$ obtained by solving **P4** satisfies

$$\begin{aligned} ||\Delta\dot{\mathbf{x}}_i|| = O\Big(\max\{\max_{j \in \mathcal{A}(\mathbf{x}_i)} \Big|\frac{\lambda_i^j}{\lambda_i^j + \Delta\bar{\lambda}_i^j} - 1\Big| ||\Delta\mathbf{x}_i||, \\ ||\Delta\mathbf{x}_i||^2\}\Big) = o(||\Delta\mathbf{x}_i||) \end{aligned} \tag{25}$$

Proof: The proof can be referred to in the Appendix. ■

Armijo-type step length search. An Armijo-type arc search is then performed as follows: Given $\eta \in (0, 1)$, the step length t_i is selected as the first number in the sequence $\{1, \eta, \eta^2, ...\}$ satisfying

$$\begin{cases} U(\mathbf{x}_i + t_i\Delta\mathbf{x}_i + t_i^2\Delta\dot{\mathbf{x}}_i) \leq U(\mathbf{x}_i) + t_i\gamma u(\mathbf{x}_i)\Delta\mathbf{x}_i, \\ d_j(\mathbf{x}_i + t_i\Delta\mathbf{x}_i + t_i^2\Delta\dot{\mathbf{x}}_i) > 0, \forall j \\ d_j(\mathbf{x}_i + t_i\Delta\mathbf{x}_i + t_i^2\Delta\dot{\mathbf{x}}_i) \geq d_i(\mathbf{x}_i), \forall j \in \mathcal{V}(\mathbf{x}_i) \end{cases} \tag{26}$$

where $\gamma \in (0, 0.5)$ is a prespecified constant value. Then, according to [31], $\mathbf{x}_{i+1}$ is updated by

$$\mathbf{x}_{i+1} = \mathbf{x}_i + t_i \Delta \mathbf{x}_i + t_i^2 \Delta \dot{\mathbf{x}}_i, \tag{27}$$

while if $\mathcal{V}(\mathbf{x}) \neq \emptyset$, $\boldsymbol{\lambda}_{i+1}$ is updated by

$$\lambda_{i+1}^j = \min \Big\{ \max\{||\Delta \mathbf{x}_i||^2, \lambda_i^j + \Delta \lambda_i^j\}, \lambda_{max} \Big\}, \tag{28}$$

where $\lambda_{max} > 0$ is a pre-defined threshold. Otherwise, we directly set $\boldsymbol{\lambda}_{i+1} = \boldsymbol{\lambda}_0$.

Overall Algorithm. Finally, we summarize our algorithm in **Algorithm 1**. Lines 4-5 perform Newton iteration and terminate the iteration until $|\Delta \bar{\mathbf{x}}_0| > \epsilon$ or $i > I$. Lines 6-8 perform descent direction correction. Lines 9-14 conduct second order decsent direction correction. Lines 15-21 perform an Armijo-type arc search method to find the optimal step length.

Algorithm 1 Primal-Dual Interior-Point Algorithm

Input: Objevtive function $U(\cdot)$;
Constraints $d_j(\cdot)$;
Parameters ϵ, I, γ σ, θ, η, κ, λ_{max};

1: Initialize $i = 0$;
2: Select $\mathbf{x}_0$ from $\mathcal{X}$;
3: **while**(true)
4: Calculate $\Delta \bar{\mathbf{x}}_i$ and $\Delta \bar{\boldsymbol{\lambda}}_i$ by solving (16) with $\mathbf{x}_i$, $\boldsymbol{\lambda}_i$ and $\bar{\boldsymbol{\mu}}_i = \mathbf{0}$;
5: **if** ($|\Delta \bar{\mathbf{x}}_0| > \epsilon$ or $i > I$)
6: Calculate $\tilde{\boldsymbol{\mu}}_i$ according to (18)
7: Calculate $\Delta \bar{\mathbf{x}}_i$ and $\Delta \bar{\boldsymbol{\lambda}}_i$ by solving (16) with $\mathbf{x}_i$, $\boldsymbol{\lambda}_i$ and $\tilde{\boldsymbol{\mu}}_i$;
8: Calculte $\Delta \mathbf{x}_i$ and $\Delta \boldsymbol{\lambda}_i$ according to (19)
9: Organize $\mathcal{A}(\mathbf{x}_i)$ and $\mathcal{V}(\mathbf{x}_i)$
10: **if** ($\mathcal{A}(\mathbf{x}_i) \neq \emptyset$)
11: Calculate $\Delta \dot{\mathbf{x}}_i$ by solving **P4**
12: **else**
13: Set $\Delta \dot{\mathbf{x}}_i = \mathbf{0}$
14: **endif**
15: Calculate t_i according to (26);
16: Calculate $\mathbf{x}_{i+1}$ according to (27);
17: **if** ($\Delta \dot{\mathbf{x}}_i == \mathbf{0}$)
18: Update $\boldsymbol{\lambda}_{i+1}$ according to (28);
19: **else**
20: Set $\boldsymbol{\lambda}_{i+1} = \boldsymbol{\lambda}_0$;
21: **endif**
22: $i = i + 1$;
23: **else**
24: $\mathbf{x}^* = \mathbf{x}_i$
25: **break**
26: **endif**
27: **endwhile**

C. Analysis

We prove the optimality, convergence and complexity of **Algorithm 1** with the following theorems and discussion:

Theorem 1 (Optimality): All the accumulation points of sequence $\{\mathbf{x}_i\}$ generated by **Algorithm 1** are KT points.

Proof: The proof can be referred to in the Appendix. ■

Theorem 2 (Convergence): **Algorithm 1** achieves local quadratically convergence if **Proposition 1** holds.

Proof: The proof can be referred to in the Appendix. ■

Complexity. In the worst case, **Algorithm 1** takes I iterations to stop. In each iteration, **Algorithm 1** calculates $H(\mathbf{x})$, solves (16) twice and **P4**. The complexity of each iteration is $O(|\mathbf{x}|^3)$. Therefore, the overall complexity of **Algorithm 1** is $O(I \cdot |\mathbf{x}|^3)$.

D. Improvements

1) Unduplicated packet sampling: Since Viper assigns a sampling probability for each queue on each switch, a packet can be sampled multiple times along its routing path, which causes duplicated packet sampling. To solve this problem, we set a packet's specific bit in the TCP/IP header field to 1 whenever a packet is sampled. This mechanism ensures all appearances of this packet on subsequent switches will be identified as a duplicated sampling packet [32] [33] so that subsequent switches bypass sampling this packet. In such case, (8) can be written by

$$S_{n_i,n_j}^{z,k} = 1 - \prod_{l_{n_a,n_b}}^{\mathcal{L}} \Big(1 - I(l_{n_a,n_b}, \mathbf{P}_{n_i,n_j}^z) s_{n_a}^k \Big). \tag{29}$$

2) Visibility Ensurance: Suppose a flow with taffic volume v, tranmitting from n_i to n_j along path $\mathbf{P}_{n_i,n_j}^z$. The demoting threshold on n_i is $\boldsymbol{\alpha} = (\alpha_0, ..., \alpha_K)$. Then, we can obtain the packet number lower bound of Q_k by

$$M_k(v, \boldsymbol{\alpha}) = \frac{\max\{0, \min\{\alpha_i - \alpha_{i-1}, v - \alpha_{i-1}\}\}}{v'} \tag{30}$$

where v' denotes the maximum packet size. If sampled packets do not contain any packet of this flow, the flow is considered invisible in the samples. Consequently, we can estimate its invisible probability by

$$\Pr(v, \boldsymbol{\alpha}, \mathbf{P}_{n_i,n_j}^z) = \prod_{k=1}^{K} (1 - S_{n_i,n_j}^{z,k})^{M_k(v,\boldsymbol{\alpha})}. \tag{31}$$

Clearly, this flow is guaranteed visible if $\exists M_k(v, \boldsymbol{\alpha}) > 0$, $S_{n_i,n_j}^{z,k} = 1$. Thus, we can ensure every flow is visible by setting $S_{n_i,n_j}^{z,1} = 1, \forall n_i, n_j, z$. According to (8) and (29), such condition can be met if $\exists n_a$, $I(l_{n_a,n_b}, \mathbf{P}_{n_i,n_j}^z) s_{n_a}^1 = 1$. In other words, Viper can ensure all flows are visible if there is a switch n_a on any path $\mathbf{P}_{n_i,n_j}^z$ that $s_{n_a}^1 = 1$.

3) Quasi-Newton Iteration and Paremeter Fixing: However, we also note that **Algorithm 1** requires calculating $H(\mathbf{x})$, which leads to huge computational overhead. Consequently, we utilize a Quasi-Newton Iteration method to approximate $H(\mathbf{x})$ by simply selecting a $H(\mathbf{x})$ to ensure $H(\mathbf{x}) + J_d(\mathbf{x})^T D_d(\mathbf{x}) J_d(\mathbf{x})$ positive definite [31]. According to Section III-E, $\mathbf{q}$, $\mathbf{s}$, and $\boldsymbol{\alpha}$ exhibit concurrent influence and interact in a synergistic manner on Viper's performance. Directly computing all optimal parameters involves substantial computational overhead. Besides, configuring $\boldsymbol{\alpha}$ also necessitates additional communication mechanisms between each end host and the controller. Thus, we suggest fixing $\boldsymbol{\alpha}$ on

each end host and only configuring optimal $\mathbf{q}$, $\mathbf{s}$ by installing corresponding routing and sampling rules on switches to address the aforementioned issues.

4) Initial Point Strategy: Moreover, interior-point based methods are susceptible to $\mathbf{x}_0$ and $\boldsymbol{\lambda}_0$. Improper $\mathbf{x}_0$ and $\boldsymbol{\lambda}_0$ lead to slow or even non-convergence, resulting in high computational overhead and sub-optimal sampling policies. Therefore, we leverage a simple heuristic algorithm to address this issue. This algorithm's key idea is to take a Netwon-like step to estimate the deviation of $\mathbf{x}_0$ and $\boldsymbol{\lambda}_0$ in terms of all constraints and shift them to $\mathbf{x}'_0$ and $\boldsymbol{\lambda}'_0$ ensuring they are sufficiently positive [34]. Specifically, lines 1-3 take a Netwon-like step to estimate the deviation of $\mathbf{x}_0$ and $\boldsymbol{\lambda}_0$ regarding all constraints. Lines 4-6 shift $\mathbf{x}_0$ and $\boldsymbol{\lambda}_0$ to sufficiently positive point $\mathbf{x}'_0$ and $\boldsymbol{\lambda}'_0$.

Algorithm 2 Initial Point Strategy

Input: $\mathbf{x}_0$, $\boldsymbol{\lambda}_0$, and parameters $\delta > 0$;

1: Organize $\mathbf{d}_0 = (d_1(\mathbf{x}_0), ..., d_{|d|}(\mathbf{x}_0))$;
2: Calculate $\Delta\bar{\mathbf{x}}_0$ and $\Delta\bar{\boldsymbol{\lambda}}_0$ by solving (16) with $\mathbf{x}_0$, $\boldsymbol{\lambda}_0$ and $\bar{\boldsymbol{\mu}}_0 = \mathbf{0}$;
3: Calculate $\Delta\mathbf{d}_0 = J_d(\mathbf{x_0})^T \Delta\bar{\mathbf{x}}_0$;
4: Calculate $\mathbf{d}'_0$ with $d'^{,j}_0 = \max\{\delta, |d_j(\mathbf{x}_0) + \Delta d^j_0|\}$;
5: Calculate $\boldsymbol{\lambda}'_0$ with $\lambda'^{,j}_0 = \max\{\delta, |\lambda^j_0 + \Delta d^j_0|\}$;
6: Calculate $\mathbf{x}'_0$ by solving $\mathbf{d}'_0 = (d_1(\mathbf{x}'_0), ..., d_{|d|}(\mathbf{x}'_0))$;
7: Return $\mathbf{x}'_0$ and $\mathbf{d}'_0$;

5) Managing Dynamic Traffic: Given the dynamic nature of traffic volume distribution and flow arrival rates, Viper must adjust its parameters accordingly. However, given the relatively stable nature of traffic volume distribution over the long term, we propose adjusting parameters solely in response to substantial changes in traffic volume distribution and flow arrival rates to mitigate the control and computational overhead associated with frequent updates of sampling policies. Moreover, frequent traffic bursts with high flow arrival rates also cause frequent parameter adjustments. Moreover, frequent traffic bursts with high flow arrival rates can lead to repeated parameter adjustments. Therefore, we reserve a buffer of sampling capacity to handle such bursts and adjust parameters only after substantial and sustained changes have been observed.

V. Evaluation

In this section, we compare Viper against state-of-the-art frameworks with both small-scale mininet testbed experiments and large-scale trace simulation experiments.

A. Small-Scale Mininet Testbed Experiments

Implementation and Experimental Settings. We conducted experiments on a high-performance server with a 104-core Intel Xeon Gold 6230R 2.10 GHz CPU, 512 GB of memory, and a 2 TB hard disk. Our network simulation involved a topology with four switches and six end hosts, as depicted in Fig. 7. Each switch was an instance of Open vSwitch, managed by a Floodlight 1.1 controller utilizing the OpenFlow 1.3 protocol. End hosts operated within separate

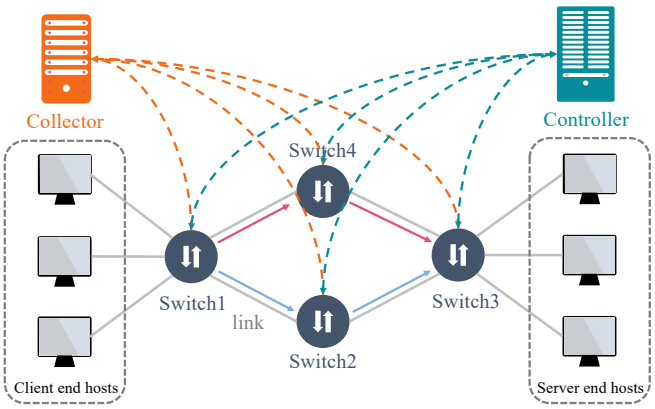


Fig. 7. Testbed experiment topology.

virtual machines, each with 16 cores, 32 GB of memory, and a 150 GB hard disk. We developed a client/server model flow generator to generate dynamic traffic reflective of real-world workloads, as shown in Fig. 8. In this model, clients initiated requests to servers to retrieve data, and servers furnished the requested data in response. The mechanisms and parameters used in our setup are detailed below:

- *Mechanisms on end hosts.* We implemented Viper's Demoting Priority Tagging Mechanism by utilizing the priority tagging module from the PIAS [35]. This mechanism operates as a kernel module between the TCP/IP stack and the Linux Traffic Control. Additionally, we enabled ECN on both end hosts and switches to realize the ECN-enabled Rate Control Mechanism.
- *Mechanisms on switches.* We employed an Open vSwitch extension [24] to realize Viper's Priority-based Per-flow Probabilistic Sampling Mechanism. Switches sampled and forwarded packets in their packet processing pipeline to a predesignated collector using UDP encapsulation. The Multi-Level Feedback Queue (MLFQ)-based Traffic Scheduling Mechanism was integrated into the switch's built-in per-port MLFQ.
- *Mechanism on the controller.* The Collaborative Control Mechanism consists of a two-path balanced routing module implemented in the Forwarding module of the Floodlight controller and and a parameter configuration module executed by a standalone Python script. The script transmitted OpenFlow Rate Modification (OFPT_RATE_MOD) messages through an OpenFlow protocol extension, supported by our modified switches [24].
- *Parameters.* We set $U(g) = \sqrt{g}$, $K = 4$, $\lambda_{max} = 10^6$, $I = 10^{-6}$, $\sigma = 2.5$, $\eta = 0.5$, $\theta = 0.5$, $\gamma = 0.2$, $C = 100$ Mbps, and $W = 150$ Kbps[2]. We selected a large enough $\mathbf{x}_0$ and initialized $\boldsymbol{\lambda}_0$ randomly.

Methods. In our comparison, we implement the following frameworks

[2]The maximum rate the switches in our setup could sustain is 50 packet per second, which is almost 150 Kbps in our experiment. Sampled packets exceed the capacity will be discarded.

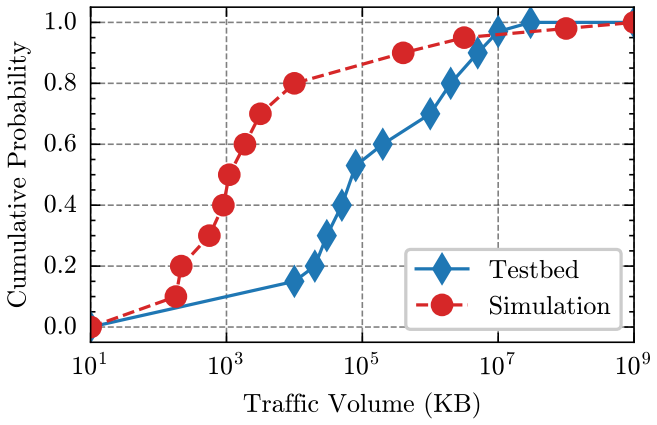


Fig. 8. Testbed and simulation traffic volume distribution.

- *Flowshark:* It employs static wildcard sampling policies for short and unclassified flows on edge switches and dynamic per-flow sampling policies for long flows.
- *SoD:* It uses a central controller to determine the optimal sampling policies for all flows based on their real-time packet rates.
- *sFlow:* In this framework, each switch adopts uniform packet sampling that samples at a $\frac{1}{X}$ rate by sampling 1 packet for X consecutive packets.

In our experiment, we implement the packet rate estimator and long flow detector of Flowshark based on the Scikit Python library. We implement the rate estimator with MLPRegressor, containing one hidden layer with 100 neurons and ReLU as the activation function. The long flow detector is implemented on the CSupport Vector Classifier (SVC) with regularization parameter $C = 5$ and the "RBF" kernel. We train these neural networks on datacenter packet traces available in [29], using features such as IP protocol, source and destination IP addresses and ports, and packet length. In our experiments, SoD also adopts the rate estimator used in Flowshark. To highlight the superior performance of Viper, we allow Flowshark to install the wildcard static sampling rules on all switches and manully configure the optimal wildcard rule sampling rate under differnt loads.

Metrics. We evaluate the performance of the above methods in terms of the following metrics:

- *Utility :* The total utility obtained for packet sampling.
- *Visibility:* The percentage of flows with at least one packet in the sampled sets.
- *Overhead:* The control overhead is the number of control messages generated for flow sampling. Computational Overhead is the running time of different methods.
- *Sampling Load:* The average and maximum normalized sampled rate on all switches.

Experimental Results. We generate 200 flows and compare Viper, FlowShark, SoD, and sFlow in terms of utility, visibility, control overhead, and sampling load under different traffic loads. Fig. 9(a) and 9(b) illustrate the visibility and utility of these methods. Viper demonstrates the highest utility (37.28% to 90.88% higher than FlowShark) and visibility (3.83% to 8.35% higher than FlowShark), FlowShark exhibits similar utility to SoD but higher visibility, while sFlow performs the worst in both metrics. Through in-depth investigation, we observe that these findings corresponded to the design principles of these methods: Viper employs flow-prioritized sampling policies, prioritizing short flows over long flows to ensure short flows are visible during the sampling process. Moreover, Viper aims to maximize sampling utility. FlowShark utilizes a similar approach by employing static wildcard sampling rules with relatively high sampling rates for short and unclassified flows. However, this method requires periodic polling flow counters to identify long flows, which involves substantial control overhead, especially with high-frequency polling. However, low-frequency polling could lead to oversampling long flows, thus compromising visibility. Besides, FlowShark aims to sample a pre-defined fraction of each flow with a minimum maximized switch sampling load, thus compromising utility. SoD dynamically adjusts the sampling rate for each flow based on its packet rate aiming to maximize utility. However, the flow rate estimation error significantly impacts its performance, causing sub-optimal sampling policies. sFlow adopts UPS, a port-based method, neglecting the flow's traffic volume and packet rate diversity. Hence, it suffers from the invisibility problem mentioned previously.

Fig. 9(c) further depicts the control overhead of all methods across various traffic loads. sFlow, being an switch built-in module, does not incur any control overhead. Viper shows a consistent and significantly lower control overhead than FlowShark and SoD, especially at low traffic loads. This phenomenon is due to Viper's implementation of flow-prioritized sampling policies, which obviates the need for periodic polling of flow counters. While, as traffic loads increase, the control overhead for FlowShark and SoD decreases. This phenomenon contributes to the fact that we send a fixed number of 200 flows. Therefore, higher traffic loads result in shorter experimental lifetimes, leading to fewer flow counter messages. In addition, FlowShark's control overhead is slightly lower than SoD's, which aligns with their design principles: SoD uses per-flow sampling policies, while FlowShark employs wildcard sampling for short and unclassified flows. Fig. 9(d) further validates our conclusion by illustrating different methods' per-second control message numbers. We further emphasize that Viper can reduce the control overhead to a negligible degree by pre-installing priority-based wildcard sampling rules.

Fig. 9(e) demonstrates the average sampling load. Viper exhibits a significantly lower average sampling load than SoD, FlowShark, and sFlow. Specifically, the average sampling load of Viper ranges from 24.59% to 38.66% lower than SoD, from 20.51% to 50.99% lower than FlowShark, and from 43.36% to 66.57% lower than sFlow. Fig. 9(f) the normalized sampling load on each switch when the traffic load is 60%. We further calculate the variance of the normalized sampling load on each switch. The statistical analysis shows Viper (1.84×10^{-5}) distributes the sampling load more evenly among switches compared to FlowShark (0.0248), SoD (0.047), and sFlow (0.003). In summary, Viper provides high visibility, low control overhead, and a reduced sampling load.

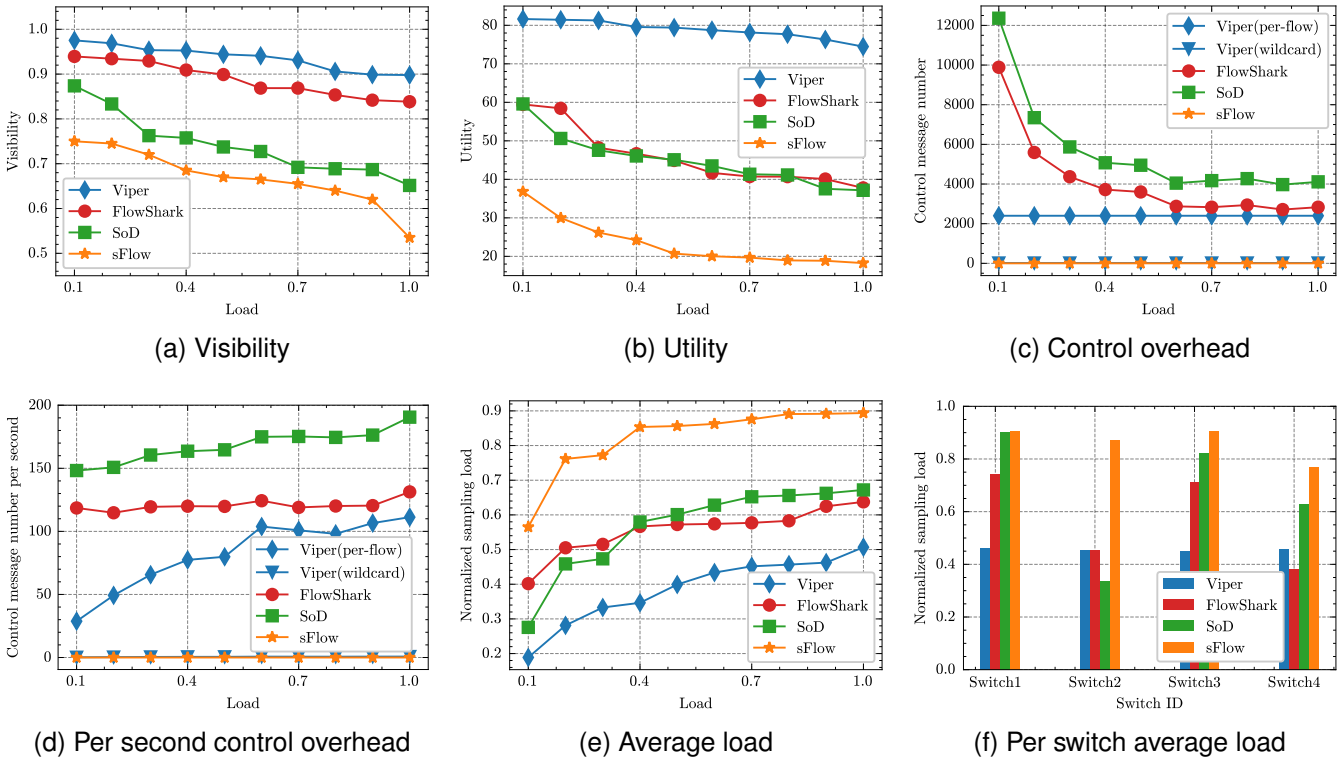


Fig. 9. Experimental results.

B. Large-Scale Trace Simulation Experiments

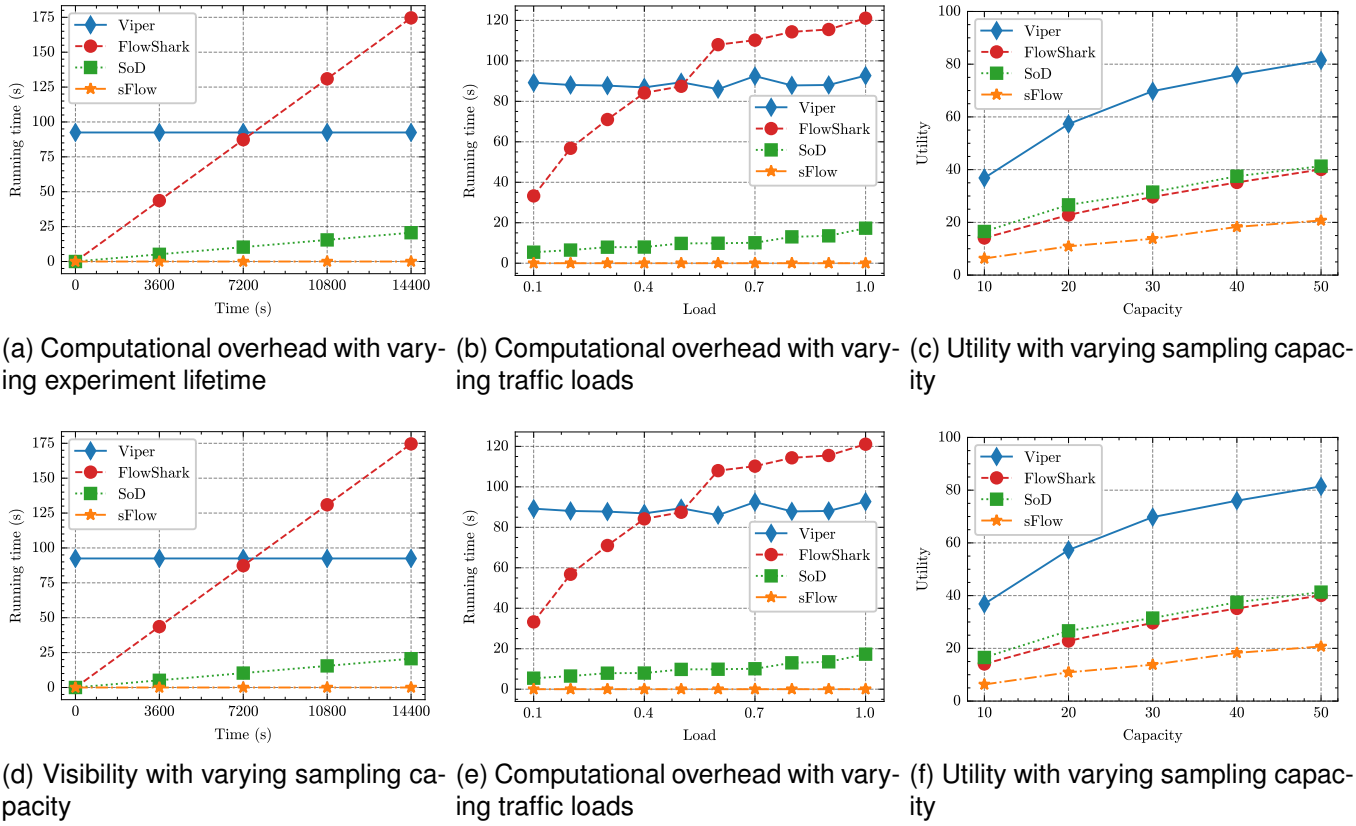
Experimental Settings. We also implemented a simulator in Python 3.8 on a workstation with 40 cores Intel Core 13100 3.60 GHz CPU, 128 GB of memory, and a 2 TB hard disk. The simulator simulate above methods by running their samlping algorithm on pcap files captured by Wireshark on each switches. The simulated network topology is same as our testbed experiment. The traffic volume distirbution of experiment trace is shown in Fig. 8.

Experimental Results Figures 10(a) and 10(b) illustrate the computational overhead associated with different methods. Initially, we set the traffic load at 60% and examine how the computational overhead varies with the experiment's lifetime. sFlow exhibits no computational overhead because it is integrated into switches as a built-in module. We are surprised to observe that FlowShark exhibits a significantly higher computational overhead than SoD. In-depth analysis reveals that this is primarily due to the adoption of long flow detection module. As the the experiment lifetime increases, the running time for FlowShark and SoD increases. In contrast, Viper's running time remains constant as its sampling policies are pre-calculated. Consider the fact that the flow arrival rate and traffic volume distribution across the entire network remain relatively stable over the long term. Consequently, Viper does not require frequent updates to its sampling policies, leading us to conclude that the associated computational overhead is acceptable. Although burst traffic is a concern, we have devised certain strategies to address this, which are detailed in our technical report. Subsequently, we set the experiment's lifetime at 3600 seconds and investigate the computational overhead under varying traffic loads. The running time for FlowShark and SoD increases with the traffic load, whereas Viper's running time remains largely unchanged. Given the heavy load and substantial flow volume typical of modern data center networks, Viper demonstrates efficient performance.

VI. Related Work

Per-Port Packet Sampling. sFlow [11] is an industry-standard technology for monitoring high-speed switched networks. In packet sampling modes, sFlow samples one in every N consecutive packet and forwards the sampled packet's header encapsulated with metadata to a collector server. While in counter sampling, sFlow forwards the counter information to the collector server. cSamp [12] achieves coordination packet sampling without explicit communication between switches by allowing switches to sample flows whose hash lies within the hash range at specific rates. Although cSamp distributes the sampling load across all switches and avoids redundant sampling, it requires additional hardware and ASIIC logic on legacy switches. OpenSample [1] works as an SDN network measurement system, leveraging sFlow packet sampling functionalities working on switches and an extension of OpenFlow protocol to communicate with the controller. Magnifier [36] achieves low overhead ingress and egress inference by inferring traffic ingress and egress points using port-based sampled traffic information and validating these inferences with mirror traffic information. While [37] focuses on the sampling node placement problem. However, .

Per-Flow Statistics Sampling. Netflow [15] is a flow statistics collecting mechanism developed by Cisco, maintaining a



(a) Computational overhead with varying experiment lifetime

(b) Computational overhead with varying traffic loads

(c) Utility with varying sampling capacity

(d) Visibility with varying sampling capacity

(e) Computational overhead with varying traffic loads

(f) Utility with varying sampling capacity

Fig. 10. Simulation results for the network.

flow cache that tracks flow statistics for each flow. It increments the statistics when packets within each flow traverse through the switch and periodically reports the statistics of active and expired flows when the number of recently expired flows reaches a predetermined maximum or after 30 seconds. Sampled NetFlow [15] is an alternative mode in NetFlow designed to reduce CPU overhead by sampling packets of a flow from the first specific fraction period of an export interval to produce NetFlow records. Such methods inevitably lead to accuracy declination. In [38], the authors focus on balancing the flow counter sampling load among multiple switches to measure network traffic statistic. To reduce the memory usage of per-flow statistics measurement, sketch-based methods, e.g., SketchVisor [18] and ElasticSketch [16], are proposed to store such information on switches. Nitrosketch [19] reduces the computational overhead while maintaining the robustness of the sketch-based method by reducing the number of hash functions and arrays and adopting adaptive sampling based on packet arrival rate. SketchFlow [20] further provides almost the same sampling rate across all flows by recognizing a sketch saturation event for a flow and sampling only the triggering packets. In [32], the authors propose to sample each flow with a probability according to its size/spread by adopting an on-chip/off-chip design where the network processor chip maintains a self-adaptive sampling module to determine per-flow sampling probability and an off-chip recording module to store flow statistics. FlexMon [39] is a hybrid traffic measurement scheme on programmable switches, where large and small flows are measured with dedicated counters in the flow table and sketches on each switch, respectively. Other methods, e.g., [17] and [40], also implement sketch-based methods on programmable switches to collect per-flow statistics. However, these methods are unable to fetch packet-level information.

Per-flow Packet Sampling. In [23], the authors argue to conduct per-flow efficient byte sampling by assigning each flow a decremented sampling probability. However, this work mainly focuses on estimating the traffic volume of each flow instead of how to implement such a mechanism in production scenarios to achieve packet sampling. SDN is a networking approach that centralizes the control of network architecture through software, allowing flexible, programmable, and scalable per-flow traffic management, which facilitates the implementation of per-flow packet sampling in a programmable form. FleXight [26] and FleXam [25] proposed a per-flow sampling extension for OpenFlow that allows the controller to determine which packets should be sampled, what parts of the sampled packets should be sent and where they should be sent. In [24], the authors propose a component on the SDN controller that allows the controller to determine the sampling rate of each flow at each switch. However, this methods require determining a sampling rate for every flow on each switch along its path according to its realtime packet rate, resulting in high control and computational overhead. FlowShark [13] aims to reduce the computational overhead by conducting static pre-specified sampling for short flows but dynamic per-flow sampling for long flows. However, this

method also involves huge control overhead for periodically polling flow entry counters to estimate per-flow traffic volume and learn real-time per-flow packet transmission rate.

VII. Conclusion

This paper presents Viper, a novel priority-based, high-visibility per-flow packet sampling framework for SDNs. Its critical idea is to resort to existing priority-based network traffic scheduling methods to prioritize short flows over long flows, and a logical central controller optimizes per-flow sampling policies for packets with specific priorities, on the data and control planes respecitvely. Specifically, Viper operates by employing mechanisms implemented across three distinct network elements: End hosts apply a demoting priority tagging mechanism, which tags packets with demoting priorities according to their sent bytes, thus differentiating between short and long flows. SDN switches utilize a priority-based probabilistic sampling mechanism, sampling packets with predetermined probabilities related to their priorities. The controller coordinates these mechanisms through a collaborative control mechanism to make routing decisions and configure related parameters, including demoting thresholds and sampling probabilities. In-depth analysis indicates that the collaborative optimization performed by the controller significantly impacts Viper's performance. Consequently, we model this control process as a nonlinear optimization problem, seeking to maximize sampling utility. Then, we propose an online primal-dual interior point method to address this optimization problem and provide mathematical proof of the algorithm's convergence, optimality, and computational efficiency. Experimental results show that Viper increases visibility by 3.83% to 8.3%, with negligible control overhead and a substantial reduction in sampling load of at least 20.51%.

Acknowledgments

Appendix
Proof of Proposition 1

According to (11b) and (12a), $\forall \lambda^j \geq 0$ and $d_j(\mathbf{x}) \geq 0$ we have

$$\min_{\mathbf{x}\in\mathcal{R}^{|\mathbf{x}|}} L(\mathbf{x},\boldsymbol{\lambda}) = \min U(\mathbf{x}) - \sum_{j=1}^{|d|} \lambda_j d_j(\mathbf{x}) \leq U(\mathbf{x}). \quad (32)$$

Besides, given $\mathbf{x}^*$, $L(\mathbf{x}^*,\boldsymbol{\lambda})$ reaches its maximum value if $\forall j$, $\lambda_j d_j(\mathbf{x}^*) = 0$, we have

$$\max_{\boldsymbol{\lambda}} L(\mathbf{x}^*,\boldsymbol{\lambda}) = U(\mathbf{x}^*) \quad (33)$$

Therefore, we have,

$$\max_{\boldsymbol{\lambda}} \min_{\mathbf{x}\in\mathcal{R}^{|\mathbf{x}|}} L(\mathbf{x},\boldsymbol{\lambda}) = \min U(\mathbf{x}). \quad (34)$$

which proves the strong duality of **P3** and **P2**.

Appendix
Proof of Proposition 2

Taking (15) into (16), we have

$$\begin{cases} -H(\mathbf{x}_i)\Delta\mathbf{x}_i + J_d(\mathbf{x}_i)^T\Delta\boldsymbol{\lambda}_i = \mathbf{0} & (35a) \\ D_\lambda(\boldsymbol{\lambda}_i)J_d(\mathbf{x}_i)\Delta\mathbf{x}_i + D_d(\mathbf{x}_i)\Delta\boldsymbol{\lambda}_i = \mathbf{0} & (35b) \end{cases}$$

Multiplying both sides of (35a) by $\Delta\mathbf{x}_i$, we have

$$-\Delta\mathbf{x}_i H(\mathbf{x}_i)\Delta\mathbf{x}_i + \sum_{j=1}^{|d|} \Delta\lambda_i^j \nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i = 0 \quad (36)$$

Then, we define $\mathcal{I}(\mathbf{x}_i) = \{j | d_j(\mathbf{x}_i) = 0\}$ and $\mathcal{J}(\mathbf{x}_i) = \{j|\lambda_i^j = 0\}$. According to (35b) and the assumption that $\boldsymbol{\lambda}_i$ is a nonnegative vector with $\lambda_i^j > 0$ if $d_j(\mathbf{x}_i) = 0$, we have

$$\nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i = 0,\ \forall j \in \mathcal{I}(\mathbf{x}_i). \quad (37)$$

and

$$\Delta\lambda_i^j = 0,\ j \in \mathcal{J}(\mathbf{x}_i). \quad (38)$$

Therefore, we have

$$\sum_{j=1}^{|d|} \Delta\lambda_i^j \nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i = \sum_{\substack{j=1 \\ j\notin\mathcal{I}(\mathbf{x}_i) \\ j\notin\mathcal{J}(\mathbf{x}_i)}}^{|d|} \Delta\lambda_i^j \nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i \quad (39)$$

Multiplying both sides of (35b) with $\Delta\lambda_i^j$ and combining with (36) and (39), we have

$$\begin{aligned} \Delta\mathbf{x}_i H(\mathbf{x}_i)\Delta\mathbf{x}_i &= \sum_{j=1}^{|d|} \Delta\lambda_i^j \nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i \\ &= -\sum_{\substack{j=1 \\ j\notin\mathcal{I}(\mathbf{x}_i) \\ j\notin\mathcal{J}(\mathbf{x}_i)}}^{|d|} \frac{(\Delta\lambda_i^j)^2}{\lambda_i^j} d_j(\mathbf{x}_i) \end{aligned} \quad (40)$$

Since $H(\mathbf{x}_i)$ is positive definte and $\boldsymbol{\lambda}_i$ is a nonnegative vector, (40) holds only when $\Delta\mathbf{x}_i = \mathbf{0}$ and $\Delta\boldsymbol{\lambda}_i = 0$. Besides, all constraints of **P1** are linearly independent. Therefore $\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i)$ must be nonsingular.

Appendix
Proof of Proposition 3

Recall that $\Delta\bar{\mathbf{x}}_i$ and $\Delta\bar{\boldsymbol{\lambda}}_i$ are solutions of (16) with $\bar{\boldsymbol{\mu}}_i = \mathbf{0}$, hence, we have

$$\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i) \begin{bmatrix} \Delta\bar{\mathbf{x}}_i \\ \Delta\bar{\boldsymbol{\lambda}}_i \end{bmatrix} = \begin{bmatrix} u(\mathbf{x}_i) - J_d(\mathbf{x}_i)^T\boldsymbol{\lambda}_i \\ -D_d(\mathbf{x}_i)\boldsymbol{\lambda}_i \end{bmatrix} \quad (41)$$

which can be furhter decomposed as

$$\begin{cases} u(\mathbf{x}_i) = -H(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i + J_d(\mathbf{x}_i)^T(\boldsymbol{\lambda}_i + \Delta\bar{\boldsymbol{\lambda}}_i) & (42a) \\ D_d(\mathbf{x}_i)(\boldsymbol{\lambda}_i + \Delta\bar{\boldsymbol{\lambda}}_i) = -D_\lambda(\boldsymbol{\lambda}_i)J_d(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i & (42b) \end{cases}$$

Scalar multiplication of both side of (42a) by $\Delta\bar{\mathbf{x}}_i$ and combining (42b), we have

$$u(\mathbf{x}_i)\Delta\mathbf{x}_i = -\Delta\bar{\mathbf{x}}_i H(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i - \sum_{j=1}^{|d|} \frac{d_j(\mathbf{x}_i)}{\lambda_i^j}(\lambda_i^j + \Delta\bar{\lambda}_i^j)^2 \quad (43)$$

Since $\forall j$, $d_j(\mathbf{x}_i) \geq 0$ and $\lambda_i^j > 0$, for any $\theta \in (0,1)$, we have

$$\theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i \leq -\theta\Delta\bar{\mathbf{x}}_i H(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i. \tag{44}$$

Combining (19a) and (20), we have

$$\begin{aligned} u(\mathbf{x}_i)\Delta\mathbf{x}_i =&(1-\varphi_i)u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i + \varphi_i u(\mathbf{x}_i)\Delta\tilde{\mathbf{x}}_i \\ =&\frac{\big(\theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i - u(\mathbf{x}_i)\Delta\tilde{\mathbf{x}}_i\big)u(\mathbf{x})\Delta\bar{\mathbf{x}}_i}{u(\mathbf{x}_i)(\Delta\bar{\mathbf{x}}_i - \Delta\tilde{\mathbf{x}}_i)} \\ &+ \frac{(1-\theta)u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i u(\mathbf{x}_i)\Delta\tilde{\mathbf{x}}_i}{u(\mathbf{x}_i)(\Delta\bar{\mathbf{x}}_i - \Delta\tilde{\mathbf{x}}_i)} \\ =&\frac{\theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i u(\mathbf{x}_i)(\Delta\bar{\mathbf{x}}_i - \Delta\tilde{\mathbf{x}}_i)}{u(\mathbf{x}_i)(\Delta\bar{\mathbf{x}}_i - \Delta\tilde{\mathbf{x}}_i)} \\ =&\theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i \end{aligned} \tag{45}$$

Clearly, if $u(\mathbf{x}_i)\Delta\tilde{\mathbf{x}}_i \leq \theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i$, then $u(\mathbf{x}_i)\Delta\mathbf{x}_i = u(\mathbf{x}_i)\Delta\tilde{\mathbf{x}}_i \leq \theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i$ establish when $\varphi_i = 1$. Thus, the first case of (20) get proven. Then, if $u(\mathbf{x}_i)\Delta\tilde{\mathbf{x}}_i > \theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i$, we have $u(\mathbf{x}_i)\Delta\mathbf{x}_i = \theta u(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i$. Combining (44), (21) get proven.

Appendix
Proof of Proposition 4

Given $\mathbf{x}_i$ and $\boldsymbol{\lambda}_i$, $\Delta\bar{\mathbf{x}}_i$ and $\Delta\bar{\boldsymbol{\lambda}}_i$, and $\Delta\tilde{\mathbf{x}}_i$ and $\Delta\tilde{\boldsymbol{\lambda}}_i$ are obtained by solving (16) with $\bar{\boldsymbol{\mu}}_i = \mathbf{0}$ and $\tilde{\boldsymbol{\mu}}_i$, respectively. we have (41) and

$$\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i)\begin{bmatrix}\Delta\bar{\mathbf{x}}_i \\ \Delta\bar{\boldsymbol{\lambda}}_i\end{bmatrix} = \begin{bmatrix} u(\mathbf{x}_i) - J_d(\mathbf{x}_i)^T\boldsymbol{\lambda}_i \\ ||\Delta\bar{\mathbf{x}}_i||^\sigma\boldsymbol{\lambda}_i - D_d(\mathbf{x}_i)\boldsymbol{\lambda}_i\end{bmatrix}. \tag{46}$$

Then, according to (19a) and (19b), we have $\Delta\mathbf{x}_i - \Delta\bar{\mathbf{x}}_i = \varphi_i(\Delta\tilde{\mathbf{x}}_i - \Delta\bar{\mathbf{x}}_i)$ and $\Delta\boldsymbol{\lambda}_i - \Delta\bar{\boldsymbol{\lambda}}_i = \varphi_i(\Delta\tilde{\boldsymbol{\lambda}}_i - \Delta\bar{\boldsymbol{\lambda}}_i)$. Therefore, we have

$$\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i)\begin{bmatrix}\Delta\mathbf{x}_i - \Delta\bar{\mathbf{x}}_i \\ \Delta\boldsymbol{\lambda}_i - \Delta\bar{\boldsymbol{\lambda}}_i\end{bmatrix} = \begin{bmatrix} 0 \\ \varphi_i||\Delta\bar{\mathbf{x}}_i||^\sigma\boldsymbol{\lambda}_i\end{bmatrix}. \tag{47}$$

Since $\varphi_i \leq 1$ and $\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i)$ is nonsingular, we have

$$\begin{aligned} &||\Delta\mathbf{x}_i - \Delta\bar{\mathbf{x}}_i||^2 + ||\Delta\boldsymbol{\lambda}_i - \Delta\bar{\boldsymbol{\lambda}}_i||^2 \\ &\qquad \leq \lambda_{max}\sup\{||\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i)^{-1}||\}||\Delta\bar{\mathbf{x}}_i||^\sigma \end{aligned} \tag{48}$$

Since $||\Delta\boldsymbol{\lambda}_i - \Delta\bar{\boldsymbol{\lambda}}_i||^2 \geq 0$, we have $||\Delta\mathbf{x}_i - \Delta\bar{\mathbf{x}}_i||^2 \leq \lambda_{max}\sup\{||\Omega(\mathbf{x}_i, \boldsymbol{\lambda}_i)^{-1}||\}||\Delta\bar{\mathbf{x}}_i||^\sigma$. **Proposition 4** get proven.

Appendix
Proof of Proposition 5

Since $\Delta\dot{\mathbf{x}}_i$ is compute according to **P4**, expanding (23b) with first oder expansion, $\forall j \in \mathcal{A}(\mathbf{x}_i)$, we have

$$\begin{aligned} \nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\dot{\mathbf{x}}_i &= \xi_i - d_j(\mathbf{x}_i + \Delta\mathbf{x}_i) \\ &= \xi_i - d_j(\mathbf{x}_i) - \nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i + O(||\Delta\mathbf{x}_i||^2) \end{aligned} \tag{49}$$

Then, according to (42b), we furhter have

$$\begin{aligned} &\nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\dot{\mathbf{x}}_i = \xi_i - d_j(\mathbf{x}_i + \Delta\mathbf{x}_i) \\ &= \xi_i + (\frac{\lambda_i^j}{\lambda_i^j + \Delta\bar{\lambda}_i^j} - 1)\nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i + O(||\Delta\mathbf{x}_i||^2). \end{aligned} \tag{50}$$

Since $\sigma > 2$, $\xi_i = o(||\Delta\mathbf{x}_i||^2)$, $\nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\dot{\mathbf{x}}_i = (\frac{\lambda_i^j}{\lambda_i^j + \Delta\bar{\lambda}_i^j} - 1)\nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i + O(||\Delta\mathbf{x}_i||^2)$. Therefore, we have (25). **Proposition 5** gets proven.

Appendix
Proof of Theorem 1

Firstly, we prove that $\Delta\mathbf{x}_i$ converges to $\mathbf{0}$ for $i \to +\infty$. Suppose $H(\mathbf{x}_i)$ is positive defined. There exist a minimum solution for **P2**. According to **Proposition 3** and **P4**, $\Delta\mathbf{x}_i$ and $\Delta\dot{\mathbf{x}}_i$ are all descent directions and $u(\mathbf{x}_i)\Delta\mathbf{x}_i \leq -\theta\Delta\bar{\mathbf{x}}_i H(\mathbf{x}_i)\Delta\bar{\mathbf{x}}_i \leq -\theta\Delta\mathbf{x}_i H(\mathbf{x}_i)\Delta\mathbf{x}_i$. According to (26), we have

$$\begin{aligned} U(\mathbf{x}_i + t_i\Delta\mathbf{x}_i + t_i^2\Delta\dot{\mathbf{x}}_i) &\leq U(\mathbf{x}_i) + t_i\gamma u(\mathbf{x}_i)\Delta\mathbf{x}_i \\ &\leq U(\mathbf{x}_i) - \theta t_i\gamma\Delta\mathbf{x}_i H(\mathbf{x}_i)\Delta\mathbf{x}_i \end{aligned} \tag{51}$$

If $\Delta\mathbf{x}_i$ does not converges to $\mathbf{0}$, $\{U(\mathbf{x}_i)\}$ is a strictly monotonically decreasing sequence that cannot converge to the optimal solution. Therefore, $\Delta\mathbf{x}_i$ converges to $\mathbf{0}$ if there exist a minimum solution for **P2**.

Then, we prove every accumulation point generated by **Algorithm 1**, denoted as $\mathbf{x}^*$, is a stationary point. According to (16) and (19), we have

$$\begin{cases} u(\mathbf{x}_i) + H(\mathbf{x}_i)\Delta\mathbf{x}_i - J_d(\mathbf{x}_i)^T(\boldsymbol{\lambda}_i + \Delta\boldsymbol{\lambda}_i) = 0 & \text{(52a)} \\ D_d(\mathbf{x}_i)(\boldsymbol{\lambda}_i + \Delta\boldsymbol{\lambda}_i) + D_\lambda(\boldsymbol{\lambda}_i)J_d(\mathbf{x}_i)\Delta\mathbf{x}_i = \varphi_i\tilde{\boldsymbol{\mu}}_i & \text{(52b)} \end{cases}$$

Suppose $|\Delta\lambda_i'| = \max|\Delta\lambda_i^j|$ and $\lim_{i\to\infty}|\Delta\lambda_i'| = +\infty$. Dividing $|\Delta\lambda_i'|$ on both side of (52), we have

$$\begin{cases} \frac{1}{|\Delta\lambda_i'|}\Big(u(\mathbf{x}_i) + H(\mathbf{x}_i)\Delta\mathbf{x}_i\Big) = \sum_{j=1}^{|d|}\frac{\lambda_i^j + \Delta\lambda_i^j}{|\Delta\lambda_i'|}\nabla_{\mathbf{x}} d_j(\mathbf{x}_i) \\ \frac{d_j(\mathbf{x}_i)(\lambda_i^j + \Delta\lambda_i^j)}{|\Delta\lambda_i'|} + \frac{\lambda_i^j\nabla_{\mathbf{x}} d_j(\mathbf{x}_i)\Delta\mathbf{x}_i}{|\Delta\lambda_i'|} = \frac{\varphi_i||\Delta\bar{\mathbf{x}}_i||^\sigma\lambda_i^j}{|\Delta\lambda_i'|} \end{cases}$$

Similarly, we define the accumulation point of $\{\boldsymbol{\lambda}_i\}$ corresponding to $\mathbf{x}^*$ as $\boldsymbol{\lambda}^*$. Since $\Delta\mathbf{x}_i$ converges to $\mathbf{0}$, $\lambda_i^j \leq \lambda_{max}$, and $\lambda_{i+1}^j = \lambda_i^j + \Delta\lambda_i^j$, for $i \to +\infty$, we have

$$\begin{cases} \sum_{j=1}^{|d|}\frac{\lambda^{*,j}}{|\lambda_i'|}\nabla_{\mathbf{x}} d_j(\mathbf{x}_i) = 0, & \text{(53a)} \\ \frac{d_j(\mathbf{x}_i)\lambda^{*,j}}{|\Delta\lambda_i'|} = 0, & \text{(53b)} \end{cases}$$

Combining with (52), we obtain $u(\mathbf{x}^*) = \mathbf{0}$ and $d_j(\mathbf{x}^*)\lambda^{*,j} = 0$. Thus, $\mathbf{x}^*$ must be a stationary point.

Finally, we prove $\mathbf{x}^*$ is also a KT point by contradiction, supposing $\mathbf{x}^*$ is not a KT point. In the first case, we assume the sequence $\{\mathbf{x}_i\}$ leaves the χ-ball about $\mathbf{x}^*$, represented as $\mathcal{B}(\mathbf{x}^*, \chi) = \{\mathbf{x} : ||\mathbf{x}^* - \mathbf{x}|| \leq \chi\}$, and jumps into the χ-ball of another stationary point. According to (26), we have.

$$||\mathbf{x}_i - \mathbf{x}_{i-1}|| = t_i||\Delta\mathbf{x}_i|| + t_i^2||\Delta\dot{\mathbf{x}}_i|| \leq 2||\Delta\mathbf{x}_i||. \tag{54}$$

Recall that $\Delta\mathbf{x}_i$ converges to $\mathbf{0}$, the sequence of $\mathbf{x}_i$ cannot jump outside of $\mathcal{B}(\mathbf{x}^*, \chi)$. In the second case, we assume $\mathbf{x}^*$ is a stationary point but not a KT point. In such case, $\exists j$ that $\lambda^{*,j}d_j(\mathbf{x}^*) = 0$ and $\lambda^{*,j} < 0$. In other word, $\mathcal{V}(\mathbf{x}_i) \neq \emptyset$. However, (26) ensures that $d_j(\mathbf{x}_{i+1}) \geq d_i(\mathbf{x}_i), \forall j \in \mathcal{V}(\mathbf{x}_i) > 0$, which conflicts with our assumption. Thus, $\mathbf{x}^*$ is also a KT point.

APPENDIX
PROOF OF THEOREM 2

Organizing $\boldsymbol{\omega}_i = \left[\mathbf{x}_i, \boldsymbol{\lambda}_i\right]^T$ and $\phi(\boldsymbol{\omega}_i, \boldsymbol{\mu}) = [u(\mathbf{x}_i) - J_d(\mathbf{x}_i)^T\boldsymbol{\lambda}_i, \boldsymbol{\mu} - D_d(\mathbf{x}_i)\boldsymbol{\lambda}_i]^T$, $\Omega(\boldsymbol{\omega}_i)$ becomes the Jacobian of $\phi(\boldsymbol{\omega}_i, \boldsymbol{\mu})$ with respect to $\boldsymbol{\omega}_i$. According to (16), we have $\Delta\bar{\boldsymbol{\omega}}_i = -\Omega(\boldsymbol{\omega}_i)^{-1}\phi(\boldsymbol{\omega}_i, 0)$ and $\Delta\tilde{\boldsymbol{\omega}}_i = -\Omega(\boldsymbol{\omega}_i)^{-1}\phi(\boldsymbol{\omega}_i, \tilde{\boldsymbol{\mu}}_i)$, which futhter derive

$$\begin{aligned} &||\Delta\tilde{\boldsymbol{\omega}}_i - \Delta\bar{\boldsymbol{\omega}}_i|| = ||\Omega(\boldsymbol{\omega}_i)^{-1}\big(\phi(\boldsymbol{\omega}_i, 0) - \phi(\boldsymbol{\omega}_i, \tilde{\boldsymbol{\mu}}_i)\big)|| \\ &= O(\tilde{\boldsymbol{\mu}}_i) = O(||\Delta\bar{\mathbf{x}}_i||^{\sigma}) = o(||\Delta\bar{\mathbf{x}}_i||^2) = o(||\Delta\bar{\boldsymbol{\omega}}_i||^2). \end{aligned} \tag{55}$$

Since $|\frac{\lambda_i^j}{\lambda_i^j + \Delta\bar{\lambda}_i^j} - 1| = O(|\Delta\bar{\lambda}_i^j|) = O(||\Delta\bar{\boldsymbol{\omega}}_i||)$, according to **Proposition 5**, we have

$$||\Delta\dot{\mathbf{x}}_i|| = o(||\Delta\bar{\boldsymbol{\omega}}_i||^2). \tag{56}$$

We further define $\mathcal{B}(\boldsymbol{\omega}, \chi) = \{\boldsymbol{\omega} : ||\boldsymbol{\omega}^* - \boldsymbol{\omega}|| \leq \chi\}$ as the set of $\boldsymbol{\omega}$ with distantance to the KT point $\boldsymbol{\omega}^* = \left[\mathbf{x}^*, \boldsymbol{\lambda}^*\right]^T$ less than χ. Then, we select $\bar{\chi}$ so that $\Omega(\boldsymbol{\omega}_i)$ is nonsingular $\forall\boldsymbol{\omega} \in \mathcal{B}(\boldsymbol{\omega}, \bar{\chi})$. According to **Thoerem 1**, sequence $\{\boldsymbol{\omega}_i\} \to \boldsymbol{\omega}^*$ for $i \to \infty$, $\exists i'$ such that $\forall i > i'$, we have $\boldsymbol{\omega}_i \in \mathcal{B}(\boldsymbol{\omega}, \chi)$. Then, we discuss the convergence of sequence $\{\boldsymbol{\lambda}_i\}$ and $\{\mathbf{x}_i\}$ in $\mathcal{B}(\boldsymbol{\omega}, \chi)$, respecitvely.

For sequence $\{\boldsymbol{\lambda}_i\}$, we consider the update of λ_{i+1}^j in two different cases: For $j \in \mathcal{A}(\mathbf{x}^*)$, according to (28), $\lambda_{i+1}^j = \lambda_i^j + \Delta\lambda_i^j$. Therefore, according to (19b) and (55), we have

$$||\lambda_{i+1}^j - \lambda_i^j - \Delta\bar{\lambda}_i^j|| = o(||\Delta\bar{\boldsymbol{\omega}}_i||^2), j \in \mathcal{A}(\mathbf{x}^*). \tag{57}$$

Then, for $j \notin \mathcal{A}(\mathbf{x}^*)$, λ_{i+1}^j can be updated either by $\lambda_i^j + \Delta\lambda_i^j$ or $||\Delta\mathbf{x}_i||^2$. As mentioned in the former case, (57) is acheived if $\lambda_{i+1}^j = \lambda_i^j + \Delta\lambda_i^j$. While if $\lambda_{i+1}^j = ||\Delta\mathbf{x}_i||^2$, since $\forall j \notin \mathcal{A}(\mathbf{x}^*)$, $\lambda^{*,j} = 0$, we can obtain.

$$||\lambda_{i+1}^j - \lambda^{*,j}|| = ||\Delta\mathbf{x}_i||^2 = O(||\Delta\bar{\boldsymbol{\omega}}_i||^2), j \notin \mathcal{A}(\mathbf{x}^*). \tag{58}$$

For sequence $\{\mathbf{x}_i\}$, since $t_i = 1$ for i large enough that $\boldsymbol{\omega}_i \in \mathcal{B}(\bar{\chi})$. According to (55) and (56), we can derive

$$||\mathbf{x}_{i+1} - \mathbf{x}_i - \Delta\bar{\mathbf{x}}_i|| = o(||\Delta\bar{\boldsymbol{\omega}}_i||^2) \tag{59}$$

Accoring to (55). (56), (57), (58) and (59), the local Q-quadratic convergence get proven.

REFERENCES

[1] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. B. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity SDN," in *Proc. IEEE ICDCS 2014 Conference, Madrid, Spain, June 30 - July 3, 2014*, pp. 228–237.

[2] X. Dong, L. Zhao, X. Zhou, K. Li, D. Guo, and T. Qiu, "An online cost-efficient transmission scheme for information-agnostic traffic in inter-datacenter networks," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 202–215, 2022.

[3] M. Zaher, A. H. Alawadi, and S. Molnár, "Sieve: A flow scheduling framework in SDN based data center networks," *Comput. Commun.*, vol. 171, pp. 99–111, 2021.

[4] A. M. Bedewy, Y. Sun, S. Kompella, and N. B. Shroff, "Optimal sampling and scheduling for timely status updates in multi-source networks," *IEEE Trans. Inf. Theory*, vol. 67, no. 6, pp. 4019–4034, 2021.

[5] P. G. Kannan, N. Budhdev, R. Joshi, and M. C. Chan, "Debugging transient faults in data centers using synchronized network-wide packet histories," in *Proc.USENIX NSDI 2021 Symposium, April 12-14, 2021*, pp. 253–268.

[6] N. Hohn and D. Veitch, "Inverting sampled traffic," *IEEE/ACM Trans. Netw.*, vol. 14, no. 1, pp. 68–80, 2006.

[7] F. Krasniqi, J. Elias, J. Leguay, and A. E. C. Redondi, "End-to-end delay prediction based on traffic matrix sampling," in *Proc. IEEE INFOCOM 2020 Conference, Toronto, ON, Canada, July 6-9, 2020*, pp. 774–779.

[8] Q. Li, Y. Liu, Z. Liu, P. Zhang, and C. Pang, "Efficient forwarding anomaly detection in software-defined networks," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 11, pp. 2676–2690, 2021.

[9] T. Ha, S. Kim, N. An, J. Narantuya, C. Jeong, J. Kim, and H. Lim, "Suspicious traffic sampling for intrusion detection in software-defined networks," *Comput. Networks*, vol. 109, pp. 172–182, 2016.

[10] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Solé-Pareta, "Analysis of the impact of sampling on netflow traffic classification," *Comput. Networks*, vol. 55, no. 5, pp. 1083–1099, 2011.

[11] sFlow.org, "sflow version 5," https://sflow.org/sflow_version_5.txt, 2024.3.30.

[12] 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 *Proc. NSDI 2008 Symposium, April 16-18, 2008, San Francisco, CA, USA*, pp. 233–246.

[13] S. SadrHaghighi, M. Dolati, M. Ghaderi, and A. Khonsari, "Flowshark: Sampling for high flow visibility in sdns," in *Proc. IEEE INFOCOM 2022 Conference, London, United Kingdom, May 2-5, 2022*, pp. 160–169.

[14] J. Alikhanov, R. Jang, M. Abuhamad, D. Mohaisen, D. Nyang, and Y. Noh, "Investigating the effect of traffic sampling on machine learning-based network intrusion detection approaches," *IEEE Access*, vol. 10, pp. 5801–5823, 2022.

[15] Cisco, "Cisco ios software configuration guide - configuring nde," https://www.cisco.com/en/US/docs/general/Test/dwerblo/broken_guide/nde.pdf, 2024.3.30.

[16] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM 2018 Conference, Budapest, Hungary, August 20-25, 2018*, pp. 561–575.

[17] A. Agarwal, Z. Liu, and S. Seshan, "Heterosketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks," in *Proc.USENIX NSDI 2022 Symposium, Renton, WA, USA, April 4-6, 2022*, pp. 719–741.

[18] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proc. ACM SIGCOMM 2017 Conference, Los Angeles, CA, USA, August 21-25, 2017*, pp. 113–126.

[19] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: robust and general sketch-based monitoring in software switches," in *Proc. the ACM SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pp. 334–350.

[20] R. Jang, D. Min, S. Moon, D. Mohaisen, and D. Nyang, "Sketchflow: Per-flow systematic sampling using sketch saturation event," in *Proc. IEEE INFOCOM 2020 Conference, Toronto, ON, Canada, July 6-9, 2020*, pp. 1339–1348.

[21] L. Yu, J. Dong, L. Chen, M. Li, B. Xu, Z. Li, L. Qiao, L. Liu, B. Zhao, and C. Zhang, "PBCNN: packet bytes-based convolutional neural network for network intrusion detection," *Comput. Networks*, vol. 194, p. 108117, 2021.

[22] R. Kumar, M. Swarnkar, G. Singal, and N. Kumar, "Iot network traffic classification using machine learning algorithms: An experimental analysis," *IEEE Internet Things J.*, vol. 9, no. 2, pp. 989–1008, 2022.

[23] F. Raspall, "Efficient packet sampling for accurate traffic measurements," *Comput. Networks*, vol. 56, no. 6, pp. 1667–1684, 2012.

[24] R. Cohen and E. Moroshko, "Sampling-on-demand in SDN," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2612–2622, 2018.

[25] S. Shirali-Shahreza and Y. Ganjali, "Flexam: flexible sampling extension for monitoring and security applications in openflow," in *Proc. ACM HotSDN 2013 Workshop, Hong Kong, China, August 16, 2013*, pp. 167–168.

[26] S. Shirali-Shahreza, *FleXight: Flexible information channel for software-defined networking*. University of Toronto (Canada), 2018.

[27] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "Opentm: Traffic matrix estimator for openflow networks," in *Proc. 2010 PAM Conference, Zurich, Switzerland, April 7-9, 2010.*, pp. 201–210.

[28] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM 2010 Conference, New Delhi, India, August 30 -September 3, 2010*, pp. 63–74.

[29] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM IMC 2010, Conference, Melbourne, Australia - November 1-3, 2010*, pp. 267–280.

[30] E. R. Panier, A. L. Tits, and J. N. Herskovits, "A qp-free, globally convergent, locally superlinearly convergent algorithm for inequality constrained optimization," *Siam Journal on Control & Optimization*, vol. 26, no. 4, pp. 788–811, 1988.

[31] T. Al., A. Wachter, S. Bakhtiari, U. Tj., and L. Ct., "A primal-dual interior-point method for nonlinear programming with strong global and local convergence properties," *SIAM Journal on Optimization: A Publication of the Society for Industrial and Applied Mathematics*, no. 1, p. 14, 2003.

[32] Y. Du, H. Huang, Y. Sun, S. Chen, and G. Gao, "Self-adaptive sampling for network traffic measurement," in *Proc. IEEE INFOCOM 2021 Conference, Vancouver, BC, Canada, May 10-13, 2021*, pp. 1–10.

[33] Y.-E. Sun, H. Huang, C. Ma, S. Chen, Y. Du, and Q. Xiao, "Online spread estimation with non-duplicate sampling," in *Proc. IEEE INFOCOM 2020 Conference, Toronto, ON, Canada, 06-09, July, 2020*, pp. 2440–2448.

[34] E. M. Gertz, J. Nocedal, and A. Sartenaer, "A starting point strategy for nonlinear interior methods," *Appl. Math. Lett.*, vol. 17, no. 8, pp. 945–952, 2004.

[35] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "PIAS: practical information-agnostic flow scheduling for commodity data centers," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 1954–1967, 2017.

[36] T. Bühler, R. Jacob, I. Poese, and L. Vanbever, "Enhancing global network monitoring with magnifier," in *Proc. NSDI 2023 Symposium, Boston, MA, April 17-19, 2023*, pp. 1521–1539.

[37] S. Yoon, T. Ha, S. Kim, and H. Lim, "Scalable traffic sampling using centrality measure on software-defined networks," *IEEE Commun. Mag.*, vol. 55, no. 7, pp. 43–49, 2017.

[38] H. Xu, S. Chen, Q. Ma, and L. Huang, "Lightweight flow distribution for collaborative traffic measurement in software defined networks," in *Proc. IEEE INFOCOM 2019 Conference, Paris, France, April 29 - May 2, 2019*, pp. 1108–1116.

[39] Y. Wang, X. Wang, S. Xu, C. He, Y. Zhang, J. Ren, and S. Yu, "Flexmon: A flexible and fine-grained traffic monitor for programmable networks," *J. Netw. Comput. Appl.*, vol. 201, p. 103344, 2022.

[40] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "Sketchlib: Enabling efficient sketch-based monitoring on programmable switches," in *Proc.USENIX NSDI 2022 Symposium, Renton, WA, USA, April 4-6, 2022*, pp. 743–759.