Network-wide Measurement on Anomalies

Zirui Liu†, Jingcheng Ju†, Guanyu Qu, Wenrui Liu, Ziyun Zhang, Kaicheng Yang,
Yiqi Wang, Tong Yang, *Member, IEEE,* Jing Hu, Duohe Ma, and Bin Cui, *Fellow, IEEE*

***Abstract*—Network measurement is crucial for network maintenance, yet the contradiction between limited resources and immense traffic remains a great challenge: ideally, the monitoring overhead should not grow with the traffic scale. Towards this goal, we present AnoMon, an efficient and accurate network measurement system whose design philosophy is to first identify the flow groups containing abnormal traffic, named abnormal flow groups (AFG), via coarse-grained passive measurement at the network edge, and then monitor only the AFG traffic with fine-grained active measurement inside the network. We implement an AnoMon prototype on a programmable-switch testbed. Experiments show that AnoMon locates anomalies up to 4.0× and 5.6× more accurately than the always-on LightGuardian and Marple, with up to 3.9× smaller bandwidth overhead than the trigger-based NetSeer. All code is available on GitHub [1].**

***Index Terms*—Network Measurement, Sketch, Flow Group**

I. Introduction

Network measurement is essential for network management and maintenance. It lets operators monitor performance, identify problems, and address them [2], [3], [4], and informs network optimization [5], [6], [7], congestion control [8], network accounting [9], *etc.* To better understand a network, it is increasingly desirable to acquire fine-grained (per-flow per-hop) and diverse (packet count, queuing delay, *etc.*) measurement data [10], [11], [12].

The contradiction between limited measurement resources and immense network traffic is a long-standing challenge. As network scale and traffic keep growing, accurately recording all traffic with limited memory becomes increasingly difficult. Traditional *sampling-based* solutions [13], [14], [6], [15], [16], [17], [18] collect coarse-grained statistics by sampling packets at switches, but have questionable accuracy and may miss important events (*e.g.*, packet drops) due to inevitable under-sampling. With the development of measurement algorithms and programmable ASICs, many *sketch-based* solutions [19], [6], [20], [10], [21], [22] compactly record approximate per-flow per-hop statistics for all traffic in a data summary. Thanks to their fast processing and small memory overhead, sketches are widely used for various measurement tasks [19], [10], [20]. For network-wide anomaly diagnosis, however, the desired state is both broader and more selective: operators must retain coverage over *all* traffic because the affected flows are not known in advance, while avoiding continuous fine-grained per-flow per-hop monitoring of every normal flow. This distinction also explains why monitoring only a preselected small set of flows is insufficient: it can miss an anomaly whose affected flows fall outside that set. In a typical data center network, only a small fraction (*e.g.*, $< 0.01\%$ as reported by [16]) of traffic experiences abnormal events (packet drops, loops, *etc.*), usually from a few flow groups. Thus, the opportunity is to maintain an always-on, lightweight summary for all traffic that can reveal these groups, and allocate detailed measurement resources only after a group is identified as abnormal.

We realize this principle through a two-phase design. In the first phase, edge devices maintain compact flow-group-level summaries for all entering and leaving traffic. A central analyzer uses these summaries to identify abnormal flow groups (AFGs), where a flow group is an operator-defined set of flows sharing a property such as a subnet, path, tenant, or service. In the second phase, the analyzer installs marking rules only for confirmed AFGs, and switches collect detailed per-flow per-hop information only for their marked packets. The first phase preserves coverage without requiring advance knowledge of the affected flows; the second phase limits the memory, pipeline, and collection-bandwidth cost of detailed measurement to the small set of abnormal flows. This resource separation is necessary because line-rate processing does not provide unlimited on-chip SRAM, register width, stateful ALUs, pipeline stages, or control-plane collection capacity.

Towards the goal above, we present AnoMon (Anomaly-Monitor), an efficient network measurement system with the following three advantages. 1) AnoMon is applicable to a range of network-wide measurement tasks, including locating packet drops, locating excessive per-hop delay, detecting packet interval jitters, and tracing forwarding paths, and it is extensible to volume/security anomalies such as SYN floods (§ IV-C). 2) AnoMon decouples always-on coverage from fine-grained diagnosis: it keeps lightweight flow-group-level monitoring for *all* traffic, and activates fine-grained per-flow per-hop measurement only for abnormal flow groups. Consequently, its monitoring overhead in terms of memory, bandwidth, and computation is linear in the number of abnormal flows, rather than all flows. 3) AnoMon can acquire fine-grained measurement information. Besides common metrics (packet counts, packet intervals, *etc.*), AnoMon can monitor per-flow per-hop attribute distribution (*e.g.*, queuing delay distribution) in a lightweight manner by collecting per-flow aggregated data (rather than per-packet data like INT [23]). Here, the *per-flow per-hop attribute distribution* of a flow refers to the empirical distribution of a packet-level attribute (*e.g.*, queuing delay,

†Zirui Liu and Jingcheng Ju contributed equally to this work.

Corresponding authors: Tong Yang and Duohe Ma (e-mail: yangtong@pku.edu.cn; maduohe@iie.ac.cn).

Zirui Liu, Wenrui Liu, Ziyun Zhang, Kaicheng Yang, Tong Yang, and Bin Cui are with the State Key Laboratory of Multimedia Information Processing, School of Computer Science, Peking University, Beijing, China.

Jingcheng Ju and Duohe Ma are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China, and also with the University of Chinese Academy of Sciences, Beijing, China.

Guanyu Qu is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

Yiqi Wang is with the University of Southern Queensland, Australia.

Jing Hu is with Huawei Technologies Co., Ltd., China.

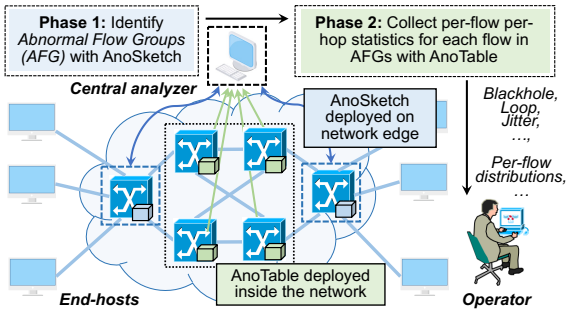


Figure 1: AnoMon overview.

packet interval, or packet size) over the packets of that flow observed at a switch within a measurement epoch; it is *not* the distribution of where packets come from, nor which path they take. This is crucial in many scenarios like attack traffic detection, SLA management, and network diagnosis (§ VI-C).

Formally, an *abnormal event* is a packet- or flow-level event relevant to diagnosis, such as a packet drop, a forwarding loop, excessive in-network latency, or an abnormal packet interval (jitter). A flow is *abnormal* if it experiences one or more abnormal events, and an *AFG* is a flow group whose aggregate abnormal signal exceeds an operator-defined threshold (formal definitions appear in § IV-A). As shown in Figure 1, AnoMon implements the two phases as follows: 1) At network edge, we propose a sketch algorithm called AnoSketch, deployed on each edge device to identify AFG. An edge device can be an edge switch or a component on end-hosts (*e.g.*, DPU, FPGA); our implementation uses an edge switch. 2) Inside the network, we propose a hash table called AnoTable, deployed on each switch to collect per-flow per-hop statistics for the flows in AFG. Both AnoSketch and AnoTable are hardware-friendly to programmable ASICs, FPGA, *etc.* A central analyzer periodically collects all AnoSketches/AnoTables network-wide in a way robust to imperfect clock synchronization, and then performs network-wide analysis. The memory overheads of AnoSketch and AnoTable are linear in the number of large flows and abnormal flows, respectively, far fewer than all flows in the network [24], [25], [26].

The **first key technique** of AnoMon, namely AnoSketch, measures flow group level statistics at network edge. Flow group level measurement grasps the macroscopic status and statistical laws of the whole network, which flow level measurement cannot obtain [19], [11], [10], [27], [20]. The key idea of AnoSketch is to sample a small amount of flows. For each sampled flow, AnoSketch reports an overestimated frequency, and for each unsampled flow, AnoSketch underestimates its frequency as zero. Given any group of flows, AnoSketch guarantees that the expected overestimation of sampled flows just offsets the expected underestimation of unsampled flows, so the estimated frequency sum is unbiased. We will theoretically and empirically show that AnoSketch is more accurate than the state-of-the-art solution [28]. We deploy AnoSketch on each edge device to measure the traffic entering and leaving the network, identifying AFG with drastic packet drops or inflated in-network latency.

The **second key technique** of AnoMon, namely AnoTable, collects fine-grained information for the traffic in AFG inside the network. Although existing systems handle many metrics (packet count, queuing delay, *etc.*) well, they all miss a crucial and challenging one: per-flow distribution. Most algorithms in the database community [29], [30], [31], [32] monitor only a single distribution. By contrast, our AnoTable provides a new feature of measuring per-flow distribution for all flows with one unified data structure. The key idea of AnoTable is named *Cell Division*: AnoTable maintains per-flow distribution in many counters. When a counter surpasses a predefined limit, indicating that its distribution is too dense, it automatically divides into multiple finer-grained counters to characterize the dense distribution more precisely. Thus AnoTable allocates more counters to complex distributions and fewer to simple ones. We will theoretically and empirically show the superiority of AnoTable over existing solutions. In AnoMon, we deploy AnoTable on each switch to collect per-flow per-hop metrics for the flows in AFG. By analyzing this information, we can pinpoint and address network problems.

We deploy AnoMon in a testbed with 10 Tofino switches and 8 end-hosts, and evaluate it with testbed experiments, large-scale simulations (NS.PY [33]), and CPU experiments. Results show that the two key components, AnoSketch and AnoTable, accurately identify AFG (up to 99% F1 score) and measure per-flow distribution, respectively, both significantly outperforming state-of-the-art solutions. As a whole, AnoMon achieves a $>$95% F1 score on locating blackholes, loops, jitters, and inflated queuing delay, which is at most 4.0$\times$ and 5.6$\times$ higher than always-on LightGuardian [10] and Marple [11]. In our testbed, AnoMon takes $<25ms$ to collect tables, identify AFG, and configure edge devices, and $<200ms$ to locate network problems. It also consumes $<$1.5Mbps extra bandwidth, up to 3.9$\times$ smaller than trigger-based NetSeer [4]. All related code is open-sourced on GitHub [1].

II. Background and Related Work

A. Measurement Tasks

This paper targets four network-wide measurement tasks.

1) Locating packet drops: Packet drops have three major causes: silent drops, blackholes, and forwarding loops. FlowRadar [34] and LossRadar [35] detect drops using a Bloom filter [36] and an Invertible Bloom Lookup Table [37]; others include NetSight [38], OmniMon [39], and more [40], [41], [42]. All treat normal and abnormal flows indiscriminately, wasting memory on the many flows that never drop packets. NetSeer [4] instead captures all dropped packets in the data plane and reports them (batched to save bandwidth) to the analyzer, achieving optimal accuracy but at higher bandwidth (§ VIII-B). Moreover, NetSeer cannot scale to other metrics, whereas metrics such as packet-size and queuing-delay distributions are pivotal for diagnosing drop causes and troubleshooting [43], [44], [45]; e.g., flows with an anomalous queuing-delay tail-quantile reveal congestion victims before they drop packets.

2) Measuring latency: Latency has two kinds: end-to-end latency and per-hop queuing delay. Most works [46], [47], [48], [49] target end-to-end latency via tailored packets. LightGuardian [10] alone acquires per-flow per-hop queuing delay,

but records all flows and thus incurs huge memory/bandwidth cost.

3) Detecting packet interval jitters: Jitter refers to drastic changes in packet interval, usually caused by queuing, congestion, or network attacks. We aim to find the flows experiencing jitters and locate the devices where they occur. Most works [50], [46], [51] detect jitters end-to-end; LightGuardian [10] is the first to detect per-flow per-hop jitters.

4) Monitoring per-flow per-hop distribution: AnoMon aims to monitor per-flow per-hop distribution for all abnormal flows. In the database community, estimating the distribution (quantiles) of a stream in limited space has been studied for decades [52]. Since exact estimation over N items requires $O(N)$ memory, many algorithms trade accuracy for space [53], [54], [55], [56], including DDSketch [29], KLL [30], and its variants [57], [31]. However, these target a single distribution, while we need a per-flow distribution for all flows; moreover, their complicated operations (*e.g.*, merging adjacent values, updating the minimum) are not pipelinable and thus cannot run on programmable ASICs. QPipe [32] runs on ASICs but measures only a single distribution, and HistSketch [58] measures per-key distributions but cannot run on ASICs.

B. Measurement Systems

Sampling-based systems: Sampling-based solutions monitor the network by sampling a traffic subset at switches. The widely-deployed NetFlow [14] and OpenSketch [6] collect flow-level information, sFlow [15] randomly samples packets, and EverFlow [16] samples each SYN packet; others mirror sampled packets to a server [38], [59] or use sketches, such as OpenSample [17], UnivMon [20], and NitroSketch [60]. Due to inevitable undersampling, these systems may miss important events and struggle to cover all anomalies.

Host-based systems: These solutions use inference algorithms to reconstruct network events at end-hosts. Many observe end-to-end flow-level performance [61], [62], [63], [64], [65]: e.g., 007 [61] "blames" each link on the paths of flows suffering TCP retransmissions, ranking links by relative drop rate. Others send probing packets, such as Netbouncer [66], Pingmesh [67], and more [68], [69], [70], [71], or leverage switches for visibility [2], [5]. Since inference cannot reconstruct all events without error, their accuracy is questionable.

Programmable switch based systems: These solutions use programmable switches to monitor anomalies inside the network, in two types. 1) *Always-on* systems constantly collect telemetry data network-wide: some collect indiscriminately, like NetSight [38], PINT [72], and more [23], [73], [10], [42], [74], [20], achieving high accuracy at high overhead, while others collect per user-defined query rules, such as Marple [11], Sonata [27], and more [75], [34], [76], [77], [78], with much lower overhead. 2) *Trigger-based* systems monitor lightweightly and collect fine-grained data only upon problem detection, reducing overhead and achieving full coverage simultaneously; typical systems include NetSeer [4], SpiderMon [12], and more [79].

Finally, existing systems usually focus on a single metric (packet loss [4], flow size [19], *etc.*), which alone cannot reconstruct an anomaly's context. AnoMon instead offers both a macroscopic (flow-group) overview and the ability to drill into various microscopic (per-flow per-hop) metrics, including the unexplored per-flow distribution.

C. Closely Related Measurements

We further discuss four closely related works. On the *algorithm* side, FlowLens [80] and SketchFeature [81] extract per-flow packet-attribute distributions (*e.g.*, packet size or inter-packet delay) in the data plane: FlowLens builds a compact quantized per-flow histogram ("flow marker") with equi-width bins for ML classifiers, while SketchFeature uses a sketch with uniformly divided bins plus a Bloom filter to extract per-flow features for in-network intrusion detection. On the *system* side, ISDC [82] proposes a serverless data-collection paradigm that distributes per-flow feature collection across switches *without* a central controller, removing the single-point bottleneck for ML-based attack detection, and CFS [83] performs cooperative, routing-oblivious network-wide flow selection, pooling the per-flow counters of all devices to maximize coverage without device coordination.

Table I positions AnoMon against these works. They mainly target *per-flow feature extraction for classification/security* (FlowLens, SketchFeature) or *network-wide counter/coverage measurement* (ISDC, CFS), whereas AnoMon targets *network-wide anomaly localization*: it couples always-on flow-group-level gating at the edge (AnoSketch) with *selective* per-flow per-hop attribute-distribution measurement inside the network (AnoTable), activated only for abnormal flow groups. Two design points distinguish AnoMon. First, AnoTable measures per-flow distributions with *geometrically* divided counters that adapt to highly skewed network attributes, more accurate than equi-width (FlowLens) and uniform (SketchFeature) binning under the same memory (§ VIII-A). Second, unlike the controller-free ISDC/CFS, AnoMon deliberately retains a lightweight analyzer for *cross-switch correlation* and *per-hop* localization, with bounded, scalable control-plane overhead (§ VIII-B). We therefore avoid claiming to be the first to measure per-flow distribution in the data plane; instead, to our knowledge, AnoMon is the first to combine flow-group-level anomaly gating with selective per-flow per-hop attribute-distribution measurement for network-wide anomaly localization.

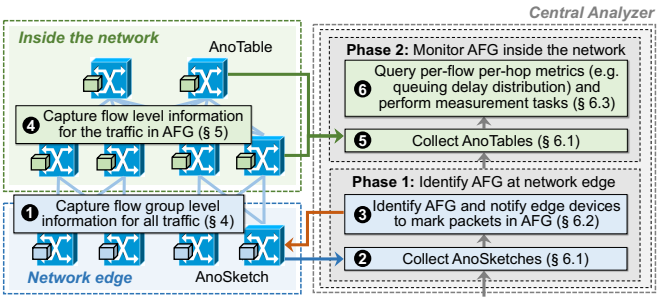


Figure 2: AnoMon workflow.

III. AnoMon Overview

As shown in Figure 2, AnoMon monitors abnormal events in two phases. In the first phase (steps 1-3), we identify abnormal flow groups (AFG) with coarse-grained passive measurement at the network edge: *1) Capture flow group level information for all traffic (§ IV):* We design a sketch called AnoSketch

Table I: Positioning AnoMon against closely related per-flow distribution and network-wide measurement works.

Work	Primary goal	Per-flow attribute distribution	Per-hop	Network-wide localization	Line-rate data plane	No central controller	Trigger/ always-on
FlowLens [80]	ML-based flow classification	✓ (equi-width)	×	×	✓	N/A	always-on
SketchFeature [81]	In-network IDS features	✓ (uniform)	×	×	✓	N/A	always-on
ISDC [82]	Distributed data collection	✓	×	∼	✓	✓	always-on
CFS [83]	Cooperative flow selection	×	×	∼	✓	✓	always-on
AnoMon (ours)	Network-wide anomaly localization	✓ (geometric)	✓	✓	✓	×	gated

to capture flow group level information, deploy it on each edge device (an edge switch in our implementation), and insert all packets entering or exiting the network into it. *2) Collect AnoSketches (§ VI-A):*A central analyzer periodically collects all AnoSketches from edge devices. We devise a mechanism to make the collection procedure robust to imperfect clock synchronization. *3) Identify AFG and notify edge devices (§ VI-B):*The analyzer queries AnoSketches to identify the flow groups experiencing abnormal events, namely AFG. After being notified by the analyzer, edge devices will mark all incoming packets in AFG to trigger active measurement inside the network.

In the second phase (steps 4-6), we monitor the traffic in AFG with fine-grained active measurement inside the network: *4) Capture flow level information for AFG traffic (§ V):*We devise a hash table called AnoTable to capture fine-grained per-flow per-hop information. Besides common metrics (packet count, maximum packet interval, *etc.*), AnoTable also supports measuring per-flow distribution for all flows. We deploy one AnoTable on each switch inside the network, and insert all marked packets into it. *5) Collect AnoTables (§ VI-A):* Similar to step 2, the central analyzer also periodically collects all AnoTables from the switches inside the network. *6) Perform network-wide measurement tasks (§ VI-C):* The central analyzer queries AnoTables to acquire per-flow per-hop metrics and perform network-wide measurement tasks. We will see that with per-flow per-hop distribution information, we can extend the usage of AnoMon to a wider range of scenarios.

IV. Monitoring at Network Edge

To grasp the macroscopic status of the network, we design AnoSketch to capture flow group level statistics by sampling large flows, and deploy it at the network edge. As AnoSketch samples only a small number of large flows, the edge module has reduced overhead. We formally define flow groups and AFG in § IV-A, present the algorithm of AnoSketch in § IV-B, and explain how to configure the edge module in § IV-C.

A. Definition of Flow Group and AFG

Flow group: A flow group refers to a set of flows $\overline{f} = \{f_1, \cdots, f_n\}$ sharing some identical property. For example, a flow group can be defined as a set of flows whose IDs (5-tuple) match partially, so that all flows coming from a specific subnet (*e.g.*, 192.168.0.0/24) form a flow group. A flow group can also be defined by other rules, such as flows with the same forwarding path or business. In AnoMon, the analyzer can group flows by any rules without reconfiguration because AnoSketch supports querying an arbitrary flow set.

Abnormal flow group (AFG): We define an AFG as a flow group in which many flows experience abnormal events. We focus on two kinds of AFG: 1) AFG with drastic Packet Drops (called PD-AFG): We define a flow group $\overline{f}$ as a PD-AFG if it loses more than $\mathcal{P}_1$ packets inside the network, where $\mathcal{P}_1$ is a predefined packet count threshold; 2) AFG with Inflated Latency (called IL-AFG): We define a flow group $\overline{f}$ as an IL-AFG if the in-network latency of more than $\mathcal{P}_2$ packets in $\overline{f}$ exceeds $\mathcal{T}$, where $\mathcal{P}_2$ is a predefined packet count threshold and $\mathcal{T}$ is a predefined latency threshold. Here, the in-network latency of a packet is the interval between its entering and exiting the network. At the edge, we only approximately find these IL-AFGs with coarse-grained measurement, so we can ignore the small clock-synchronization error across edge devices; we explain how the other modules of AnoMon handle imperfect clock synchronization in § VI-A.

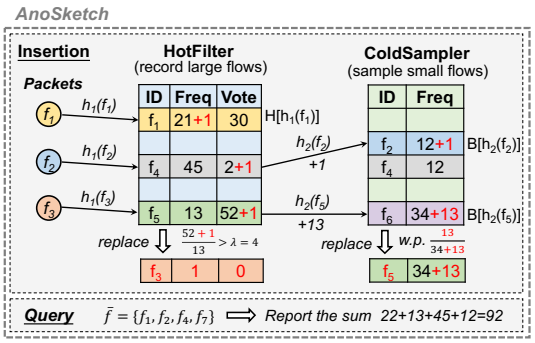


Figure 3: Data structure of AnoSketch ($\lambda = 4$). All buckets are initially empty, with the flow ID empty and the frequency/vote fields set to zero; the figure shows an intermediate state after several packet insertions.

B. The AnoSketch Algorithm

Rationale: AnoSketch aims at supporting queries to arbitrary flow groups, which can be cast as subset sum estimation: given a set of flows, each with a weight (*e.g.*, packet count), we estimate the total weight of any subset. The most advanced solutions [84], [28] sample a small number of flows: they overestimate the sampled flows and underestimate the unsampled ones, offsetting these two errors to ensure unbiased estimation for any subset. Since large flows contribute more to the subset sum, the key idea of AnoSketch is a structure called HotFilter that accurately records large flows while still sampling small flows. By allocating more memory to large flows, AnoSketch keeps the unbiased property and, as we prove theoretically, achieves smaller estimation variance than the state-of-the-art sampling solution [28]. Moreover, its estimates for large flows are accurate rather than seriously biased upward [84], [28], enabling an additional function of finding global heavy changes that guides AnoMon to identify suspected AFG (§ VI-B) and pinpoint the culprit device (§ VI-C).

Data structure (Figure 3): AnoSketch has two parts: a HotFilter recording large flows and a ColdSampler sampling small

flows. 1) HotFilter is an array of buckets $\mathcal{H}[1], \cdots, \mathcal{H}[m_1]$, each of which records *flow ID*, *frequency* (packet count), and a *vote* field used to evict small flows. 2) ColdSampler is an array of buckets $\mathcal{B}[1], \cdots, \mathcal{B}[m_2]$, each of which records *flow ID* and *frequency*. Initially, all buckets in both HotFilter and ColdSampler are empty, with the flow ID empty and the frequency/vote fields set to zero.

Insertion (pseudocode in § 6 of the supplementary material [85]): The insertion procedure takes two steps. 1) Given an incoming packet of flow f_i, we first insert it into HotFilter. We calculate hash function $h_1(f_i)$ and check whether bucket $\mathcal{H}[h_1(f_i)]$ is empty or f_i is recorded in $\mathcal{H}[h_1(f_i)]$. If so, we insert f_i into $\mathcal{H}[h_1(f_i)]$, increase $\mathcal{H}[h_1(f_i)].freq$ by one, and the insertion procedure ends. Otherwise, we increase $\mathcal{H}[h_1(f_i)].vote$ by one. If $\frac{\mathcal{H}[h_1(f_i)].vote}{\mathcal{H}[h_1(f_i)].freq} > \lambda$ (λ is a predefined threshold), meaning that there might exist a flow larger than the residing flow f_r in $\mathcal{H}[h_1(f_i)]$, we evict the residing flow and insert f_i into $\mathcal{H}[h_1(f_i)]$. We also insert the evicted flow f_r and its recorded frequency into ColdSampler. Otherwise, we just insert f_i into ColdSampler. 2) Given a pair of flow ID f_i and weight w, ColdSampler also first calculates hash function $h_2(f_i)$ to locate the hashed bucket $\mathcal{B}[h_2(f_i)]$. We add w to $\mathcal{B}[h_2(f_i)].freq$. If $\mathcal{B}[h_2(f_i)]$ is empty, we insert f_i into $\mathcal{B}[h_2(f_i)]$. Otherwise, if f_i is not recorded in $\mathcal{B}[h_2(f_i)]$, we replace the residing flow with f_i with probability $p = \frac{w}{\mathcal{B}[h_2(f_i)].freq}$.

Examples (Figure 3): For f_1 (recorded in $\mathcal{H}[h_1(f_1)]$), we just increase its *freq*. For f_2 (not in HotFilter), we increase $\mathcal{H}[h_1(f_2)].vote$ and insert $(f_2, 1)$ into ColdSampler. For f_3 (not in HotFilter), we increase the vote; since $\frac{vote}{freq} > \lambda$, we evict the residing flow f_5, replace it with f_3, and insert $(f_5, 13)$ into ColdSampler, where it replaces f_6 with probability $\frac{13}{47}$.

Query: To query a flow group $\overline{f}$, we check all buckets in HotFilter and ColdSampler, and sum up the frequency of all recorded flows in $\overline{f}$. For example, in Figure 3, to query $\overline{f} = \{f_1, f_2, f_4, f_7\}$, we sum up their recorded frequency to get $\hat{f}_1 + \hat{f}_2 + \hat{f}_4 + \hat{f}_7 = 92$. As f_4 is recorded in both HotFilter and ColdSampler, its estimated frequency $\hat{f}_4$ is the sum of the frequencies recorded in these two components. As f_7 is not recorded in AnoSketch, its estimated frequency $\hat{f}_7$ is 0.

Mathematical analysis: We theoretically prove that AnoSketch achieves unbiased estimation and smaller estimation variance than the state-of-the-art sampling solution [28].

Theorem IV.1. *For any flow f_i, suppose its real frequency is $F(f_i)$ and its estimated frequency is $\hat{F}(f_i)$, we prove that $\hat{F}(f_i)$ is unbiased, namely $\mathbb{E}\left[\hat{F}(f_i)\right] = F(f_i)$.*

Remark. We prove this theorem using mathematical induction in § 1 of our supplementary material [85]. This theorem claims that for any flow, its recorded frequency in AnoSketch is unbiased, and thus for any flow group, its estimated sum is also unbiased. Intuitively, AnoSketch overestimates the sampled flows and underestimates the unsampled ones, and the expected errors offset each other, yielding unbiasedness. Although the error for a single flow might be large, for a flow group with both sampled and unsampled flows the overall variance is acceptable.

Theorem IV.2. *For any flow f_i, AnoSketch has smaller estimation variance $Var\left[\hat{F}(f_i)\right]$ than CocoSketch [28].*

Remark. We prove this theorem from the aspect of variance increment in § 1 of our supplementary material [85]. This theorem claims that AnoSketch achieves higher accuracy in estimating subset sums than the state-of-the-art solution. We will validate this conclusion with experiments in § VIII-A.

C. Configuration of Edge Devices

We describe how to configure an edge device to perform flow group level measurement with AnoSketch, and explain how to further query the collected AnoSketches to identify abnormal large flows and AFG in § VI-B.

Configuration for identifying PD-AFG: To identify the AFG with drastic packet drops, we deploy two AnoSketches on each edge device: $AnoSketch_{in}$ and $AnoSketch_{out}$. For each packet entering/exiting the network, we insert it into $AnoSketch_{in}$/$AnoSketch_{out}$ at the edge device.

Configuration for identifying IL-AFG: To identify the AFG with inflated in-network latency, we deploy one AnoSketch on each edge switch: $AnoSketch_{lat}$. We configure each edge switch to insert the current timestamp t_{in} into the header of each packet entering the network via INT [23], [86]. For each packet exiting the network, the edge switch acquires the current timestamp t_{out}, and calculates the in-network latency of this packet $\Delta t = t_{out} - t_{in}$. If $\Delta t > \mathcal{T}$, where $\mathcal{T}$ is the predefined latency threshold of IL-AFG (§ IV-A), the edge switch inserts this packet into $AnoSketch_{lat}$.

Discussion: We finally discuss the necessity of AnoSketch (or a sampling-based solution) over simple solutions like allocating a static counter to each flow group. The foremost benefit is that our solution lets the analyzer group flows by any rules without pre-configuring edge devices: by observing the common characteristics of the sampled flows experiencing abnormal events (*e.g.*, sharing the same forwarding link), the analyzer first identifies some suspected AFGs, and then pinpoints the real AFGs by querying them in the AnoSketches (§ VI-B).

Extending AnoSketch to other anomaly signals: AnoSketch is not limited to packet-drop or latency signals; as an unbiased subset-sum estimator of the packet/event count of arbitrary flow groups, it can be configured to identify other volume- or security-related anomalies. For example, to detect SYN-flood-like anomalies, the edge device inserts only the SYN packets into a dedicated AnoSketch, and the analyzer queries flow groups defined by destination prefix, destination service, or ingress-egress pair; a destination group whose SYN count (or SYN-to-ACK imbalance) exceeds an operator-defined threshold is identified as an AFG and subsequently monitored by AnoTable. We illustrate this extensibility with a SYN-flood case study in § VIII-A (Figure 6c). Our main evaluation focuses on packet drops and latency-related anomalies, and we do not claim AnoMon to be a complete DDoS/IDS system, as it does not perform payload- or signature-level analysis.

V. Monitoring Inside the Network

To collect fine-grained information for AFG inside the network, we design AnoTable to capture per-flow per-hop

aggregated statistics, and deploy it on each switch inside the network. Beyond common metrics, AnoTable captures per-flow per-hop attribute distributions (*e.g.*, queuing delay or packet interval) for the flows in AFG. While prior data-plane feature extractors such as FlowLens [80] and SketchFeature [81] also measure per-flow attribute distributions, AnoTable divides its counters *geometrically* to match the skewed nature of network attributes, and is integrated into a network-wide gating-then-localization workflow (§ II-C). As only AFG packets are inserted into AnoTable, the inside-network module has reduced overhead that scales only with the abnormal traffic. We present the basic structure of AnoTable in § V-A, extend AnoTable to monitor per-flow distribution in § V-B, and explain how to configure the inside-network module in § V-C.

A. Basic Structure of AnoTable

To closely monitor the traffic in AFG, a straightforward solution is to mirror all AFG packets to the analyzer, but this strawman incurs significant bandwidth overhead. We instead aggregate the measurement information into a hash table (called AnoTable) on the data plane, which operators can flexibly configure for various metrics, and periodically send the aggregated per-flow per-hop statistics to the analyzer.

Data structure (top of Figure 4): AnoTable has d bucket arrays $\mathcal{D}_1, \cdots, \mathcal{D}_d$. Each array $\mathcal{D}_j$ has a hash function $g_j(\cdot)$ mapping flows into a bucket. Each bucket records a flow ID and some user-defined metrics. We offer two types of counters: a *sum* counter and a *maximum* counter. Operators can flexibly arrange these counters to record various metrics like packet/byte count, last arrival time, average/maximum packet interval, *etc.*

Insertion (pseudocode in § 6 of the supplementary material [85]): Given an incoming packet p_i of flow f_i (in AFG), we check its hashed buckets in the d arrays. If a hashed bucket $\mathcal{D}_j[g_j(f_i)]$ is empty or stores f_i, we update this bucket and the insertion procedure ends. For example, suppose each bucket stores a packet count (cnt) and a last arrival time ($time$), we set $\mathcal{D}_j[g_j(f_i)].ID$ to f_i if $\mathcal{D}_j[g_j(f_i)]$ is empty, increase $\mathcal{D}_j[g_j(f_i)].cnt$ by one, and update $\mathcal{D}_j[g_j(f_i)].time$ to the current time. If all the d hashed buckets are full, we evict the residing flow in the last hashed bucket $\mathcal{D}_d[g_d(f_i)]$ to make room for f_i, and report the evicted flow and its recorded metrics to the analyzer. We could instead evict the least recent (LRU) or least frequent (LFU) flow, but these require the switch recirculation mechanism; we choose the current design for simplicity.

Examples (top of Figure 4): The AnoTable here has $d = 2$ arrays storing, for each flow, the packet count (*sum* counter), the last arrival time (*maximum* counter), and a queuing-delay histogram (§ V-B). For f_1, its first hashed bucket $\mathcal{D}_1[g_1(f_1)]$ is empty, so we insert f_1 into it. For f_2, it already resides in $\mathcal{D}_2[g_2(f_2)]$, so we increase its *cnt* and update its *time* to 350. For f_3, both hashed buckets are occupied, so we evict the residing flow f_6 in the last bucket to insert f_3, and send the evicted flow and its metrics to the analyzer.

B. Capturing Per-flow Distribution

Motivation: Although the *sum* and *maximum* counters in basic AnoTable satisfy most tasks, some require the finer-grained per-flow per-hop distribution. We present three use cases. *1) SLA Management:* Insights into the per-flow per-hop queuing delay and packet interval distribution let operators identify potential service-quality deviations and act to maintain consistent quality. *2) Attack Detection:* Analyzing the per-flow per-hop distribution of packet sizes and intervals helps detect malicious traffic or security breaches, which often display distinct characteristics such as high entropy [45]. *3) Network Diagnosis:* The per-flow per-hop distribution aids diagnosis. For instance, a congested flow may show an initial increase in queuing delay followed by packet drops; by identifying flows with anomalous delay distributions, preemptive measures such as rerouting or rate-limiting can be applied before packet loss.

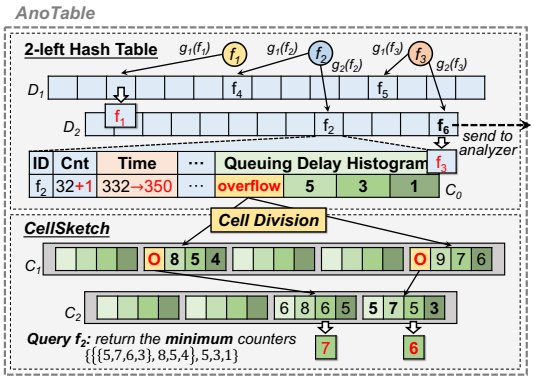


Figure 4: Data structure of AnoTable and CellSketch ($d = 2$, $r = 4$, $v = 2$, $L = 2$).

As discussed in § II-A, existing solutions only measure the distribution of a single flow [32], [29], [30], [31], whereas our goal is the per-flow distribution for all flows. Building one separate data structure per flow is inefficient, as it allocates uniform memory to all distributions, whereas simple ones can be characterized with less memory. An ideal solution should maintain the distributions of all flows in one unified structure that dynamically allocates more memory to complex distributions and less memory to simple ones, and it should also avoid circular dependency so as to be deployed on programmable ASICs.

Problem statement: We first formalize the problem of per-flow distribution estimation. Consider a multiset of n items $\mathcal{S} = \{s_1, \cdots, s_n\}$ with $s_1 \leqslant s_2 \leqslant \cdots \leqslant s_n$, the p-quantile of $\mathcal{S}$ is defined as $Q_p := s_{\lfloor p(n-1) \rfloor + 1}$. Estimating the distribution of $\mathcal{S}$ refers to reporting the p-quantile $\hat{Q}_p$ for $\forall p \in [0, 1]$. In the networking context, each flow f corresponds to one multiset $\mathcal{S}_f$, where each item s_i represents the predefined attribute of a packet of f, which can take many forms (*e.g.*, queuing delay, packet interval, *etc.*) as specified by operators for particular tasks. We aim at measuring the distribution for any flow.

Rationale: Towards the above goal, we introduce an extended module to AnoTable called CellSketch. The key idea is to maintain the per-flow distribution of all flows in many counters organized hierarchically. For each flow, we first maintain its coarse-grained distribution with a few counters at the first layer; when a counter's value surpasses a predetermined threshold, it automatically divides into several finer-grained counters at the next layer. Through this process akin to *cell division*, complex distributions are recorded in more counters and simple ones in a few. Although some works in other contexts also dynamically

adapt their shapes to network status [87], [88], their memory usage is usually not fixed and must be frequently adjusted with complicated operations; by contrast, our solution maintains fixed memory usage and is free of circular dependency.

Data structure (bottom of Figure 4): Below we explain how to enable AnoTable to measure per-flow queuing delay distribution. We introduce a basic unit called *histogram*, which consists of r counters, each recording the number of packets whose attribute value falls between a certain interval (*e.g.*, 0∼20 μs). We add one queuing delay histogram $\mathcal{C}_0$ to each bucket of AnoTable, and introduce another structure called CellSketch, a hierarchical structure with L histogram arrays $\mathcal{C}_1, \cdots, \mathcal{C}_L$. If a counter in the bucket of AnoTable overflows, it will automatically divide itself into v histograms in the first layer of CellSketch, and we call this counter the parent counter of its v child histograms. The r counters in each child histogram form a finer-grained distribution of its parent counter. For example, for a parent counter with the interval 0∼15 μs, the intervals of its four child counters are 0∼1 μs, 1∼3 μs, 3∼7 μs, and 7∼15 μs. We configure the interval lengths of the r counters within each histogram to follow a geometric sequence rather than a uniform one, because packet-level attributes such as queuing delay and packet interval span multiple orders of magnitude and are highly skewed (most packets have small values, a few very large) [89]. With uniform intervals, the counters either give insufficient resolution in the dense small-value region or waste counters in the sparse large-value region. Geometric intervals instead bound the *relative* (rather than absolute) error across scales, matching quantile-based diagnosis and consistent with relative-error quantile sketches such as DDSketch [29]. As we prove in Theorem V.1 and validate in § VIII-A, the ratio α of the geometric sequence trades accuracy against memory: a smaller α yields finer resolution and lower relative error but requires more counters; we use $\alpha = 2$ by default. Similarly, if a counter in the l_{th} ($l \geqslant 1$) array of CellSketch overflows, it will divide itself into one histogram in the $(l+1)_{th}$ array.

Insertion (pseudocode in § 6 of the supplementary material [85]): Given a packet of flow f_i, we first acquire its queuing delay t_i. We check the counter whose interval contains t_i (denoted as $\mathcal{C}_0[t_i]$) in the histogram in AnoTable. If $\mathcal{C}_0[t_i]$ does not overflow, we increase $\mathcal{C}_0[t_i]$ by one and the insertion procedure ends. Otherwise, we divide $\mathcal{C}_0[t_i]$ into v histograms in CellSketch. Similarly, if the counter in CellSketch overflows, we proceed to divide it into a histogram in the next array. This procedure repeats until no counter overflows or we reach the last array. We use the combination of flow ID f_i and counter ID (address) $\&\mathcal{C}_l[h][t_i]$ as the hashed key to locate the histogram in CellSketch. This resolves two issues: if we used only the counter ID, all flows colliding into the same histogram in $\mathcal{C}_l$ would continue to collide in $\mathcal{C}_{l+1}$; if we used only the flow ID, all overflowed counters in the v hashed histograms in $\mathcal{C}_l$ would accumulate to the same histogram in $\mathcal{C}_{l+1}$.

Query (pseudocode in § 6 of the supplementary material [85]): To query the delay distribution of flow f_i, we first check its delay histogram in AnoTable. For each overflowed counter in the histogram, we further query its child histograms in CellSketch to acquire finer-grained distribution. As multiple flows can collide into the same histogram in CellSketch, we obtain the result by taking the minimum value at each corresponding position from the v child histograms. Similarly, when querying CellSketch, an overflowed counter is further resolved by querying its child histogram in the next array.

Example (bottom of Figure 4): We use an example to illustrate the insertion and query procedure of CellSketch, where each histogram has $r = 4$ counters, each counter in AnoTable divides into $v = 2$ child histograms in CellSketch, and CellSketch has $L = 2$ arrays. Let the lengths of the r intervals in each histogram form a geometric sequence with a ratio of 2, and let the intervals of the $r = 4$ counters in $\mathcal{C}_0$ be $[0, 225]$, $[225, 675]$, $[675, 1575]$, $[1575, 3375]$. *1) Insertion:* For the packet of f_2, suppose its queuing delay is 5 μs, which falls in the first counter in $\mathcal{C}_0$. As this counter overflows, we check its $v = 2$ child histograms in $\mathcal{C}_1$. The queuing delay (5 μs) falls in the first counter, and both counters in $\mathcal{C}_1$ overflow again. In $\mathcal{C}_2$, the queuing delay falls in the third counter with range $[3, 7]$, and these two counters do not overflow. Therefore, we increase each of them by one. *2) Query:* To query the delay distribution of f_2, we first check its histogram in $\mathcal{C}_0$. As $\mathcal{C}_0[1]$ overflows, we further query its $v = 2$ child histograms in CellSketch. As both child histograms in $\mathcal{C}_1$ have an overflowed counter, we further query $\mathcal{C}_2$ to acquire the finer-grained distribution. The $v = 2$ query results in CellSketch are $\{\{6, 8, 7, 5\}, 8, 5, 4\}$ and $\{\{5, 7, 6, 3\}, 9, 7, 6\}$. After taking the minimum counters, we get the finer-grained distribution of $\mathcal{C}_0[1]$ as $\{\{5, 7, 6, 3\}, 8, 5, 4\}$. In this way, we get the final distribution as $\{\{\{5, 7, 6, 3\}, 8, 5, 4\}, 5, 3, 1\}$.

Mathematical analysis: We theoretically analyze the error upper bound of CellSketch in the following theorem.

Theorem V.1. *Consider a histogram $\mathcal{C}_l[h]$ in CellSketch with r counters whose interval lengths follow a geometric sequence with a ratio of α. Let x be the real p-quantile ($0 < p < 1$), and suppose x falls into $\mathcal{C}_l[h][k]$. We have the relative error $\mathcal{E} := \frac{|x-\hat{x}|}{x}$ is bounded by $\mathcal{E} \leqslant \frac{\alpha^k - \alpha^{k-1}}{\alpha^k + \alpha^{k-1} - 2}$.*

Remark. We will prove this theorem in § 1 of our supplementary material [85]. This theorem gives an error upper bound for a single histogram; by repeatedly applying it to the child histograms, we can get finer-grained error bounds for CellSketch. A smaller α goes with a smaller error, but uses more counters to divide the initial interval, leading to more memory usage; in practice, we recommend setting $\alpha = 2$.

C. Configuration of Inside-network Devices

We configure inside-network switches for fine-grained measurement with AnoTable, and query the collected AnoTables to perform various measurement tasks (§ VI-C). 1) For locating packet drops, we deploy a basic AnoTable on each switch and add a packet-count field (sum counter) to each bucket. 2) For locating inflated queuing delay, we deploy a AnoTable (with CellSketch) in each switch's egress pipeline and add a queuing-delay histogram to each bucket; for each packet in AFG, we acquire its queuing delay in the ingress pipeline and insert it into the histogram. 3) For locating packet interval jitters, we deploy a AnoTable (with CellSketch) on each switch and add a timestamp field ($maximum$ counter, recording each flow's last

arrival time) and a packet-interval histogram to each bucket; for each packet, we compute the interval from this timestamp and the current time and insert it into the histogram. 4) We can also add histograms of other user-defined attributes to extend AnoMon (*e.g.*, diagnosing the causes of packet drops).

VI. Collection and Analysis

We first describe how the central analyzer collects AnoSketch/AnoTable and deals with imperfect clock synchronization in § VI-A, then how to identify AFG with AnoSketches and perform measurement tasks with AnoTables in § VI-B and § VI-C. In this section, we collectively refer to AnoSketch and AnoTable as "tables".

A. Collecting AnoSketches and AnoTables

To avoid collision between packet insertion and table collection, we divide the timeline into consecutive 0/1 epochs and build two copies for each table on the data plane: an active and an idle table. In each epoch, the active table records information while the idle table is collected by the analyzer and then reinitialized. Below we elaborate on the collection process and explain how to make it compatible with unsynchronized clocks.

Table collection with epoch-bit: On each switch, we maintain a periodically flipped 0/1 timestamp, called epoch-bit, which divides the timeline into consecutive fixed-length epochs. The central analyzer also maintains this epoch-bit and periodically synchronizes its clock with each switch. The two table copies on each switch each correspond to one epoch-bit value and monitor the epochs with that value. At the edge device, for each packet entering the network, we obtain its 0/1 epoch-bit from the current time and write it into the packet header; at each subsequent switch on the path, we insert the packet into the table corresponding to the epoch-bit in its header. Whenever the local epoch-bit flips, the analyzer collects all tables monitoring the previous epoch, and each switch switches to the other table for the next epoch. After collecting all tables, the analyzer identifies AFG and performs measurement tasks with them. We will see that the total time taken by table collection, AFG identification, and edge device configuration is far smaller than epoch length (§ VIII-C).

Postponing collection to address clock asynchronization: The above procedure has two issues: 1) If the analyzer collects tables immediately after the local epoch-bit flips, many packets carrying the previous epoch-bit are still inside the network and not yet inserted into the previous-epoch tables, so they are incorrectly judged as packet drops. 2) In practice, the clocks of different devices are hard to synchronize perfectly [90], [91], introducing error into our results. Both issues are addressed by postponing collection: whenever the local epoch-bit flips, the analyzer waits an additional period before collecting tables. The waiting time should exceed the maximum packet transmission time plus the maximum clock difference, guaranteeing that all packets with the previous epoch-bit have exited the network and eliminating the consequences of asynchronous clocks.

Timestamp and tag lifecycle: We summarize how AnoMon uses timestamps and packet tags, and to what extent it relies on clock synchronization. *(1) Timestamps.* The per-hop queuing delay used by AnoTable is computed locally within a single switch (from the ingress/egress timestamps of the same device), and is therefore *insensitive* to inter-switch clock skew. Only the edge-to-edge in-network latency $\Delta t = t_{out} - t_{in}$ used for IL-AFG detection depends on clock consistency between the ingress and egress edge devices; this requires only *bounded* skew (not nanosecond-level global synchronization), which we tolerate via the waiting interval before table collection. NTP suffices in our testbed; data centers with PTP would only tighten the bound. *(2) Epoch-bit lifecycle.* The ingress edge sets the epoch-bit; each on-path switch writes the packet into the table selected by this bit; the analyzer waits and then collects; once a packet leaves the measurement domain, its epoch-bit is no longer used. *(3) AFG-bit lifecycle.* The ingress edge sets the AFG-bit for marked flows; in-network switches read it to decide whether to insert the packet into AnoTable; the *egress edge clears the AFG-bit (and any in-domain telemetry tag) before forwarding the packet outside the measurement domain*, so that no tag leaks to end hosts or external networks. Our prototype carries these two bits in reserved TCP-header bits for simplicity; a production deployment can instead use INT/IOAM shim headers, tunnel/encapsulation headers, VLAN/MPLS metadata, or switch-local metadata, without affecting the algorithm.

Control-plane overhead and scalability: We analyze the control-plane overhead of the central analyzer, which is the main scalability concern for a network-wide system. Let N_e and N_s be the number of edge and total switches, S_{sketch} and S_{table} the size of one AnoSketch and one AnoTable, E the epoch length, N_{afg} the number of confirmed AFGs, and S_{rule} the size of one marking rule. The per-epoch *collection* bandwidth is $B_{collect} = (N_e S_{sketch} + N_s S_{table})/E$, the *notification* bandwidth is $B_{notify} = (N_e N_{afg} S_{rule})/E$, and the per-edge marking-table memory is $M_{mark} = N_{afg} \cdot S_{rule}$. The central *analysis* time is $O(N_e B_{sketch} + N_s B_{table} + Q)$, where B_{sketch}/B_{table} are the bucket counts and Q is the number of candidate-group queries. A key property is that the analyzer processes *aggregated per-epoch tables*, not per-packet records; hence its overhead grows only with the number of switches, the table size, and the number of AFGs, and is *independent of the traffic volume*. Moreover, in hierarchical data-center topologies (*e.g.*, fat-tree, leaf-spine), the analyzer can collect tables *layer by layer* following the topology depth, reducing the instantaneous collection burst and aligning collection with packet residence time. We empirically evaluate this overhead at different topology scales in § VIII-B.

Detection latency and short-lived anomalies: Because AnoMon collects tables periodically, there is an inherent delay between an anomaly occurring and being diagnosed. The worst-case latency to *detect* an AFG is $T_{detect} \leq E + T^{sketch}_{collect} + T^{sketch}_{analysis} + T_{config}$, and to obtain its fine-grained per-hop *diagnosis* is $T_{diagnose} \leq T_{detect} + E + T^{table}_{collect} + T^{table}_{analysis}$, *i.e.*, the fine-grained measurement of a first-time AFG is activated at least one epoch after detection. Consequently, a persistent anomaly is fully diagnosed within a couple of epochs, but an anomaly shorter than one epoch may have disappeared before the in-network measurement is triggered; in that case AnoMon can still record the aggregate evidence at the edge AnoSketch (and thus detect the AFG), but may not capture the complete

per-hop distribution. AnoMon therefore trades instantaneous full-flow fine-grained visibility for scalable, diagnosis-oriented measurement, and best suits group-correlated or recurring anomalies. In our simulation, anomalies lasting at least half an epoch are detected with F1 ≥ 0.8, whereas shorter anomalies are detected less reliably, consistent with the bound above. For operators requiring sub-epoch response, AnoMon can additionally support a *fast-trigger* mode, in which an edge device that observes a threshold-crossing sketch counter (or an end-host report) immediately triggers AFG marking instead of waiting for the next periodic collection, trading extra control-plane messages for lower latency. We evaluate this mode in § VIII-B, where it captures anomalies as short as $\sim$200 ms and cuts the detection latency by about $20\times$.

B. Identifying Abnormal Flow Groups

To identify AFG, the analyzer first checks the collected AnoSketches to find a partial set $\mathcal{F}_{abn}$ of abnormal flows. In our implementation, $\mathcal{F}_{abn}$ contains the sampled flows experiencing packet drops (global heavy changes) or inflated in-network latency, and can also include flows of applications with end-to-end performance degradation reported by users. The analyzer then *dynamically generates* candidate flow groups from $\mathcal{F}_{abn}$ rather than relying on a pre-configured list. It aggregates the flows in $\mathcal{F}_{abn}$ along several candidate dimensions (*e.g.*, source/destination prefix, ingress/egress edge, forwarding path, tenant/service) to produce candidate groups; for each, it queries the collected AnoSketches for the group's aggregate abnormal signal, and confirms a real AFG only if the signal exceeds the corresponding threshold ($\mathcal{P}_1$/$\mathcal{P}_2$/$\mathcal{T}$ in § IV-A). Thus AnoMon does *not* require edge devices to pre-install all AFG rules: operators only specify the candidate grouping dimensions and thresholds, the actual AFGs are *generated and verified online* by the analyzer (enabled by the subset-sum query capability of AnoSketch), and only the confirmed AFGs are pushed to edge devices as marking rules. Finally, the analyzer notifies all edge devices to mark the traffic of confirmed AFGs to trigger active measurement inside the network. Below we elaborate on the process of finding global heavy changes and identifying the two kinds of AFG.

Finding global heavy changes: We first retrieve the IDs of all flows recorded in $AnoSketch_{in}$. For each flow ID f_i, we query it in $AnoSketch_{in}$/$AnoSketch_{out}$ to get its number of packets entering/exiting the network n_{in}/n_{out}. If the packet drop $\Delta n = n_{in} - n_{out}$ exceeds a predefined threshold, we report f_i as a global heavy change and add it to $\mathcal{F}_{abn}$.

Identifying PD-AFG: Given a suspected AFG $\overline{f}$, we query all $AnoSketch_{in}$/$AnoSketch_{out}$ to get its number of packets entering/exiting the network n_{in}/n_{out}. We calculate the number of dropped packets $\Delta n = n_{in} - n_{out}$ and report $\overline{f}$ as PD-AFG if $\Delta n > \mathcal{P}_1$, where $\mathcal{P}_1$ is the threshold in § IV-A.

Identifying IL-AFG: Given a suspected AFG $\overline{f}$, we query all $AnoSketch_{lat}$ to get its number of packets n whose in-network latency is larger than $\mathcal{T}$. We report $\overline{f}$ as IL-AFG if $n > \mathcal{P}_2$, where $\mathcal{T}$ and $\mathcal{P}_2$ are thresholds defined in § IV-A.

C. Performing Measurement Tasks

The analyzer takes two steps to perform network-wide measurement tasks. First, it constructs a query set $\mathcal{F}_q$ of the abnormal flows in AFG, which may contain the sampled global heavy changes in § VI-B, the flows with end-to-end incidents reported by the operators (*e.g.*, TCP duplicate ACKs, TCP timeouts), and even some small flows of a certain business of interest. Second, for each flow f_i in $\mathcal{F}_q$, the analyzer queries it in the AnoTables on the switches along the forwarding path of f_i to locate the specific culprit device or link. For each task in § II-A, the analyzer localizes the culprit hop from the per-hop statistics of the victim flow. For packet drops, a slight per-hop packet-count decrease, a sudden drop to 0, or a sudden increase localizes random drops, a blackhole, or a forwarding loop, respectively. For inflated queuing delay, an inflated per-hop delay quantile (*e.g.*, the 0.8-quantile exceeding $100\mu s$) localizes the culprit switch. For jitters, a rapidly changing per-hop packet-interval quantile (*e.g.*, a suddenly increased 0.9-quantile) localizes the culprit switch. Finally, querying the AnoTables (with CellSketch) yields the distribution of any user-defined attribute at per-flow per-hop granularity, extending AnoMon to network diagnosis and troubleshooting (§ V-B). The detailed per-task procedures and more distribution-estimation examples are provided in § 2 and § 3.4 of our supplementary material [85].

VII. Implementation

We have implemented AnoMon in a testbed built with 10 Edgecore Wedge 100BF-32X switches (with Tofino ASIC) [92] and 8 end-hosts in the FatTree topology shown in Figure 2. We implement AnoSketch and AnoTable (with CellSketch) on the programmable data plane using the P4 language [93], and deploy them on the edge switch and the inside-network switch in a fully pipelined manner without using the mirror or recirculate mechanisms. On the analyzer, we implement three modules in a unified DPDK framework: table collection; table analysis (identifying AFG and performing the four measurement tasks); and notification-packet sending. For space, the detailed data-plane pipeline layouts, the per-switch hardware resource usage, and the optimized table-collection procedure are deferred to § 3 of our supplementary material [85]; below we highlight the two aspects most relevant to deployment feasibility: the per-edge marking memory and the on-chip bucket layout.

Marking packets: In AnoMon, the edge switch stamps each packet with the following two marks. 1) AFG mark: After the analyzer notifies each edge switch of the identified AFG, the edge switch marks the packet in AFG before sending it into the network. 2) Epoch mark: Recall that to avoid collision between packet insertion and sketch collection, we build two copies of AnoSketch on each edge switch: an *active* sketch and an *idle* sketch. These two sketches are used to record information and are collected by the analyzer in turn (§ VI-A). Thus, each packet needs an epoch mark indicating which copy it should be inserted into. In our implementation, we reserve two bits in the TCP header of each packet: one AFG-bit and one epoch-bit. As soon as a packet enters the switch, we calculate the epoch-bit according to the current time, and mark the epoch-bit in its header (Stage 0). Afterwards, we check whether the packet is in AFG, and mark its AFG-bit if so (Stage 1). The AFG membership is decided by a match-action table installed by

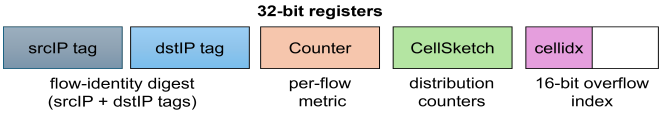


Figure 5: AnoTable bucket layout on Tofino registers.

the analyzer, whose entries are the confirmed AFG rules (*e.g.*, prefix or path matches). This marking table is the only per-switch memory introduced by the notification: its size is the number of confirmed AFGs (N_{afg}) times the per-entry width, which is small in practice. In our testbed, the marking table is provisioned for up to 5000 AFG rules, occupying only a few tens of KB of TCAM/SRAM; the analyzer can further compress it by aggregating prefixes, merging overlapping AFGs, and installing rules only at ingress edge switches.

AnoTable bucket layout on Tofino: A natural concern is whether an AnoTable bucket—a flow identity plus several metrics—fits the limited bit-width of a register cell on a programmable ASIC. The key point is that a bucket is *not* one wide cell: each field is a separate $\leq$ 32-bit register array, all addressed by the same d-way hash index and accessed in different pipeline stages (Figure 5, Table II). To save width, the flow identity is a compact digest (a 32-bit source-IP tag plus a 32-bit destination-IP tag) rather than the full 5-tuple; a bucket collision only affects accuracy (mitigated by d=2-way hashing), *never* forwarding correctness. The 18-bit queuing delay is mapped to a 2-bit offset selecting one of the r=4 geometric sub-counters, and an overflowed counter is redirected to a CellSketch child histogram via a 16-bit bucket-to-cell index; we keep two ping-pong copies of all arrays for epoch toggling. With $B_{bucket} = B_{id} + B_{cnt} + B_{time} + r \cdot B_{hist}$, a register memory of M bits holds about M/B_{bucket} buckets (up to 32768 per array in our implementation). This confirms that AnoTable is realizable under the register-width constraints of programmable ASICs.

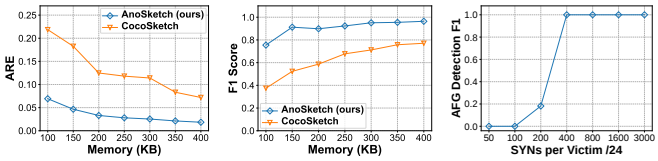
Table II: AnoTable register layout on Tofino (digest key, d=2 ways, ping-pong $\times$2).

Field	Width	#entries	Role
Counter (metric)	32 bit	32768	per-flow metric counter
Source-IP tag	32 bit	16384	flow-identity digest
Destination-IP tag	32 bit	16384	flow-identity digest
CellSketch counter	32 bit	32768	distribution sub-counters
Bucket-to-cell index	16 bit	32768	overflow redirection

VIII. Experimental Results

We extensively evaluate AnoMon with CPU experiments (§ VIII-A), NS.PY [33] simulations (§ VIII-B), and testbed experiments (§ VIII-C). The following results are highlighted.

- **AnoSketch and AnoTable outperform prior art.** In CPU experiments, AnoSketch estimates flow-group sizes >3.15× more accurately than CocoSketch [28], and AnoTable (with CellSketch) estimates per-flow distributions with >5.5× smaller error than DDSketch [29], KLL [30], GK [94], and HistSketch [58].
- **AnoMon accurately identifies abnormal flow groups and locates anomalies.** In simulations, AnoMon achieves >90% F1 on identifying AFG and >95% F1 on locating four kinds of anomalies, up to 4.0×/5.6× better than LightGuardian [10] and Marple [11].
- **AnoMon has small response time and lightweight overhead.** In our testbed, AnoMon takes <25ms to collect AnoSketches, identify AFG, and reconfigure edge devices, and <200ms to collect AnoTables and locate anomalies, with extra bandwidth always <1.5Mbps; in simulation, its bandwidth overhead is up to 3.9× smaller than NetSeer [4].



(a) Flow-group ARE. (b) Heavy-change F1. (c) SYN-flood F1.

Figure 6: Accuracy and extensibility of AnoSketch *vs.* CocoSketch: (a) flow-group size ARE; (b) F1 of finding global heavy changes; (c) SYN-flood AFG detection F1 *vs.* flood intensity.

Metrics: We report the recall rate (RR), precision rate (PR), and F_1 score for AFG identification and anomaly localization; the average relative error (ARE) [19] for flow-group size estimation; and the average logarithm error (ALE) [95], which adapts to skewed real-world attributes, for per-flow distribution estimation. Their formal definitions are given in § 4 of our supplementary material [85].

Default setting: By default, we set the monitoring epoch to 5 seconds. For AnoSketch, we set the memory ratio of HotFilter to ColdSampler to 1 : 3. For AnoTable (with CellSketch), we set the memory ratio of the Hash Table to CellSketch as 1 : 2. We set $d = 3, r = 8, L = 4$, and use 10-bit attribute counters (with the overflow threshold of 1024).

Parameter guidelines: Table III summarizes the parameters of AnoMon, their default values, and how to set them. We distinguish two kinds of parameters. *Operator-policy* parameters (the epoch length E and the AFG thresholds $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{T}$) encode the operator's monitoring intent: a larger E reduces collection overhead but increases detection latency, while larger thresholds reduce the number of reported AFGs (higher precision, lower recall). *Data-structure* parameters (d, r, L, α, and the two memory ratios) trade accuracy against on-chip memory: larger d/r/L and smaller α improve accuracy at the cost of more memory and pipeline stages. We study the sensitivity of the most important parameters across our evaluation: the per-switch memory (Figure 10a), the anomaly intensity (§ 5 of the supplementary material [85]), the epoch length (Figure 10f), and the geometric ratio α (Figure 8b).

Table III: Parameters of AnoMon, their defaults, and configuration guidelines.

Param	Meaning	Default	Type
E	Monitoring epoch length	5 s	policy
$\mathcal{P}_1$	Packet-drop threshold (PD-AFG)	—	policy
$\mathcal{P}_2, \mathcal{T}$	Latency thresholds (IL-AFG)	—	policy
HF:CS	HotFilter:ColdSampler mem. ratio	1:3	data-struct
HT:CS	HashTable:CellSketch mem. ratio	1:2	data-struct
d	# bucket arrays of AnoTable	3	data-struct
r	# counters per histogram	8	data-struct
L	# CellSketch levels	4	data-struct
α	Geometric interval ratio	2	data-struct
T_o	Counter overflow threshold	1024	data-struct

A. CPU Experiments

We conduct CPU experiments to evaluate AnoSketch and AnoTable using CAIDA [96] , a 1-minute trace collected from backbone links with 27M packets from 1.3M flows.

Experiments on AnoSketch: We evaluate the performance of AnoSketch on measuring flow groups and finding abnormal large flows (global heavy changes). We compare AnoSketch with CocoSketch [28], a state-of-the-art algorithm designed for measuring subset sum on programmable switches.

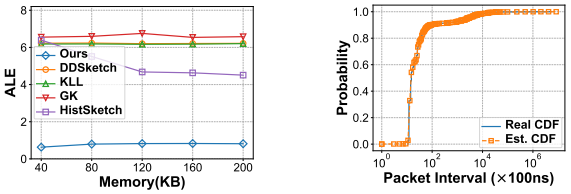
Accuracy on measuring flow group size (Figure 6a): We find AnoSketch achieves 3.15×-4.49× better accuracy than CocoSketch. We use the source IP /24 prefix to divide the 1.3M flows in CAIDA into 250K flow groups (source IP subnets), and report the ARE on measuring the size of the top-300 flow groups. The results show that AnoSketch achieves 3.15×-4.49× smaller ARE than CocoSketch. For example, when using 150KB memory, the ARE of AnoSketch and CocoSketch is 0.047 and 0.182.

Accuracy on finding global heavy changes (Figure 6b): We find AnoSketch achieves $1.25\times-2.01\times$ better F1 score than CocoSketch. We divide the trace into two windows, and use a method similar to that described in § VI-B to find those flows whose frequency change between the two windows exceeds 5K. The results show that AnoSketch achieves $19.3\%-38.9\%$ higher F1 score on finding global heavy changes than CocoSketch. For example, when using 150KB memory, the F1 score of AnoSketch and CocoSketch is 52.3% and 91.2%.

Extensibility to SYN floods (Figure 6c): We demonstrate that AnoSketch is a general arbitrary-flow-set event-count estimator through a SYN-flood case study, where the edge inserts only SYN packets into a dedicated AnoSketch and the analyzer flags as a SYN-flood AFG any destination /24 group whose estimated SYN count exceeds an operator threshold. We inject floods of varying intensity (SYNs per victim destination) on top of normal SYN traffic, and report the detection F1. AnoSketch reliably identifies the victim groups once the flood intensity exceeds the threshold (F1 reaches 1.0 at $\geq$400 SYNs/victim in our setting), with no false positives on normal groups, while correctly leaving sub-threshold groups unflagged. This confirms that the same subset-sum primitive used for PD-/IL-AFGs extends to volume/security anomalies; a full DDoS/IDS pipeline (*e.g.*, payload analysis) is beyond our scope.

Experiments on AnoTable: We evaluate the performance of AnoTable (with CellSketch) on estimating per-flow attribute distribution, where we use *packet interval* as the attribute. We compare AnoTable with four state-of-the-art algorithms from the database community: HistSketch [58], DDSketch [29], KLL [30], and GK [94]. Note that the latter three algorithms can only estimate the distribution of a single flow, so we build multiple copies of their data structures and use a hash function to assign each flow to one copy.

Accuracy comparison with prior art (Figure 7a): We find that AnoTable achieves >5.5× smaller error than prior art. For example, when using 200KB memory, the ALE of AnoTable is 0.82, while those of DDSketch, KLL, GK, and HistSketch are 6.21, 6.20, 6.58, and 4.50. AnoTable has smaller errors because it can automatically allocate more memory to complex



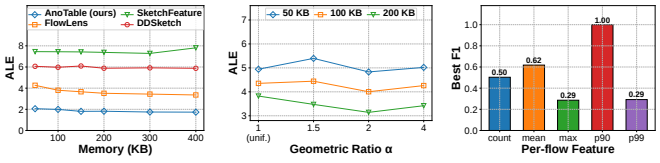
(a) Comparison of average error (ALE) with prior art. (b) Per-flow distribution estimation example.

Figure 7: Accuracy of AnoTable (with CellSketch).

distributions and less memory to simple ones, thereby adapting well to the skewed distribution of real-world data. By contrast, DDSketch, KLL, and GK can only estimate the distribution of a single flow. When directly applied to estimating per-flow distribution of all flows, they allocate the same memory to every flow. However, the simple distribution of most small flows can be characterized with less memory. Although HistSketch separates large flows from small flows and allocates more (but the same) memory to each large flow, its accuracy is still significantly worse than that of AnoTable. This is because HistSketch only coarsely categorizes flows into large and small flows, whereas AnoTable finely allocates different amounts of space to different distributions, which better utilizes the space.

Comparison with FlowLens and SketchFeature (Figure 8a): We further compare AnoTable with the two data-plane per-flow distribution extractors discussed in § II-C, FlowLens [80] and SketchFeature [81], on the same per-flow packet-interval distribution task under equal memory, with each baseline tuned to its best configuration. AnoTable achieves about $1.9-2.1\times$ lower ALE than FlowLens and about $4.2\times$ lower than SketchFeature (DDSketch is also shown for reference). The reason is structural: FlowLens quantizes attributes into *equi-width* bins and SketchFeature divides the attribute range into *uniform* levels, so both waste resolution on the dense small-value region of the heavy-tailed interval distribution, whereas AnoTable's geometrically divided counters bound the relative error across scales and adaptively refine only the dense regions via cell division.

Impact of the geometric ratio α (Figure 8b): We study how the geometric interval ratio α affects the per-flow distribution estimation error. We vary α at several memory budgets, where α=1 corresponds to uniform binning. We find that the error first decreases and then increases as α grows, reaching a minimum at α=2: a too-small α approaches uniform binning and wastes resolution in the sparse large-value region, while a too-large α makes the high-value bins too coarse. At 200KB, α=2 reduces the error by about $1.2\times$ compared with uniform binning. This



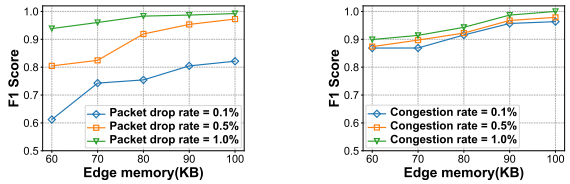
(a) ALE *vs.* memory. (b) Vary ratio α. (c) Dist. *vs.* scalar.

Figure 8: Additional evaluation of AnoTable (with CellSketch): (a) ALE of per-flow 0.95-quantile estimation *vs.* memory; (b) estimation error *vs.* the geometric interval ratio α; (c) best F1 of detecting tail-inflated flows.

matches Theorem V.1, which shows that α controls the relative error, and justifies our default of α=2 as a balance between accuracy and memory.

Estimation examples (Figure 7b): We demonstrate the effectiveness of AnoTable in estimating packet interval distribution of the largest flow in CAIDA. The results show that the estimated CDF closely fits the real CDF, showing that AnoTable can accurately characterize per-flow distribution. We provide more distribution estimation examples of AnoTable in § 2 of the supplementary material [85].

Benefit of distribution over scalar counters (Figure 8c): We finally show why capturing the per-flow *distribution* is necessary and not redundant with scalar counters. We model a transient congestion that inflates the bulk tail of a flow's queuing-delay distribution, while rare extreme spikes are shared by all flows and the mean is only mildly affected. We then detect the affected flows using different per-flow features and report the best achievable F1. Under a subtle anomaly, the distribution quantile $p90$ attains F1 $= 1.0$, whereas packet count, mean, and maximum attain only 0.50, 0.62, and 0.29; even the extreme quantile $p99$ fails (0.29) because it is dominated by the rare spikes common to all flows. This confirms that the diagnosis-relevant signal lives in a specific region of the distribution that scalar counters cannot recover, justifying AnoTable's per-flow per-hop distribution measurement.



(a) PD-AFG (b) IL-AFG

Figure 9: Performance of AnoMon on identifying AFG.

B. Simulation Experiments

We conduct large-scale, network-wide simulations using NS.PY [33] to evaluate the accuracy of AnoMon on identifying AFG (PD-AFG, IL-AFG) and locating network anomalies (blackholes, loops, jitters, and inflated queuing delay). We build an 8-ary standard FatTree topology with 80 switches and 128 end-hosts. We generate 24M packets of 0.1M TCP flows following DCTCP [97]. The source and destination of each flow are chosen uniformly at random. We divide flows into flow groups according to their ECMP forwarding paths. We randomly select 4% flow groups, and let some flows in them proactively experience anomalies at some switches/links. We create network anomalies according to a Poisson process with an intensity of 0.1 events per second. The duration of each anomaly follows $N(5, 2.5)$. We compare AnoMon with three measurement systems: LightGuardian [10], Marple [11], and NetSeer [4]. We reimplemented these systems in NS.PY. For Marple, we only use its on-chip hash table and do not use its off-chip backing store (Redis database). As discussed in § II-B, LightGuardian [10] and Marple [11] are two always-on solutions that collect data with sketches and hash tables on the programmable data plane. They can support various tasks by collecting different statistics. NetSeer [4] is a trigger-based solution specifically designed for detecting packet drops on programmable switches. By default, we set the monitoring epoch to 5 seconds, set the anomaly rate to 0.1%, and set per-switch memory to 100KB. For the tasks of blackholes and loops, we use PD-AFG, and for the tasks of jitters and inflated queuing delay, we use IL-AFG.

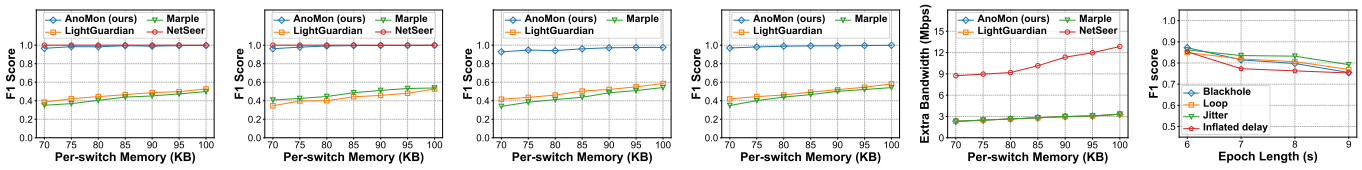
Baseline configuration and fairness: To make the comparison fair, all systems are run under the *same* traffic trace, anomaly injection, per-switch memory budget, and monitoring epoch, and we tune each baseline to its best-effort configuration following its paper. We also clarify the task coverage of each baseline rather than forcing a comparison on unsupported tasks. LightGuardian and Marple are always-on systems that can support all four tasks by collecting the corresponding statistics, so we compare them on every task. NetSeer is a trigger-based system specialized for packet drops: it captures all dropped packets in the data plane and thus attains optimal packet-drop accuracy, but it does not natively provide the distributional metrics required by jitter or inflated-delay diagnosis. Therefore, we compare NetSeer only on the packet-drop-related tasks (with its bandwidth overhead shown in Figure 10e) and mark the other tasks as *not applicable*, instead of reporting a misleadingly low score for it. For Marple, we use only its on-chip hash table and not its off-chip backing store (Redis), because all the compared systems are constrained to the same on-chip memory budget; using off-chip memory would change the resource model and make the comparison unfair.

Performance on identifying AFG (Figure 9): We find that AnoMon achieves up to 99% F1 score on identifying PD-AFG and IL-AFG. In Figure 9a/9b, we vary the memory of AnoSketch on each edge switch and the *packet drop rate/congestion rate* at each culprit switch. We can see that a larger anomaly rate goes with a higher F1 score, as it makes the anomaly more pronounced. Under 100KB memory, AnoMon achieves >99% F1 score under the anomaly rate of 1%.

Performance on locating network anomalies (Figure 10):

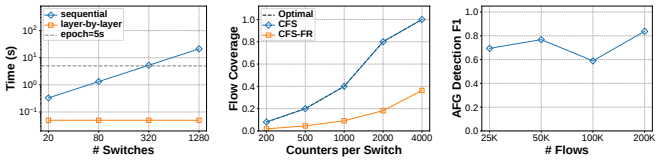
Accuracy on locating anomalies (Figure 10a-10d): AnoMon achieves up to 99% F1 score on locating blackholes, loops, and inflated queuing delays, and 98% on locating jitters, which is up to $2.0\times-2.7\times$ higher than LightGuardian and Marple. As the corresponding anomaly rate (packet-drop, loop, jitter, or congestion rate) drops, the F1 score of AnoMon decreases only slightly, whereas those of LightGuardian and Marple decrease drastically (a full anomaly-intensity sweep is reported in § 5 of the supplementary material [85]); for example, under a 0.05% packet-drop rate AnoMon still exceeds 0.8 F1 while LightGuardian and Marple fall to 0.52 and 0.48. As NetSeer captures all dropped packets in the data plane (§ II-A), its F1 score is always 1 on the drop-related tasks (blackholes and loops), but it does not provide the distributional metrics required by the jitter and delay tasks.

5) Bandwidth overhead (Figure 10e): The bandwidth overhead of AnoMon is $3.4\times$-$3.9\times$ smaller than trigger-based NetSeer, and is almost the same as LightGuardian and Marple. This is because NetSeer needs to report all event packets to the analyzer, whereas the other solutions only periodically report the aggregated measurement statistics in sketches/hash tables, thereby significantly reducing bandwidth overhead.



(a) Blackhole. (b) Loop. (c) Jitter. (d) Inflated delay. (e) BW overhead. (f) Epoch length.

Figure 10: Network-wide performance of AnoMon on locating anomalies: F_1 *vs.* per-switch memory for the four anomaly types (a–d), bandwidth overhead (e), and epoch-length sensitivity (f). F_1 *vs.* anomaly intensity is in § 5 of the supplementary material [85].



(a) Analyze time. (b) CFS coverage. (c) Flow-scale F1.

Figure 11: Scalability and system-level comparison.

Summary: AnoMon is more accurate than always-on LightGuardian and Marple, because they struggle to maintain the information of all flows in the data plane with limited memory. By contrast, AnoMon only records the information of a few abnormal flows, enabling small overhead and high accuracy. Although trigger-based NetSeer has the best accuracy on packet drop tasks (blackhole and loop), it incurs large bandwidth overhead and cannot generalize to other tasks.

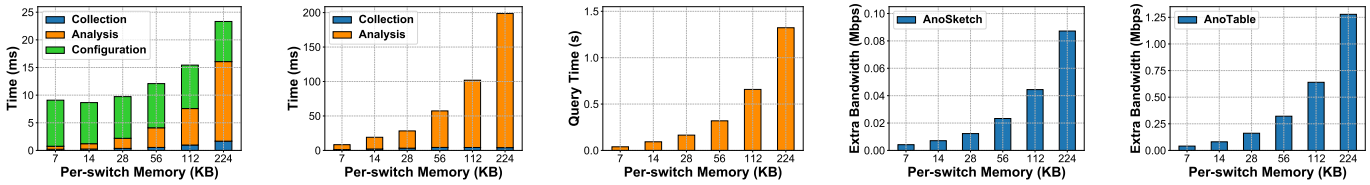
Impact of epoch length (Figure 10f): We find that a smaller epoch length goes with a higher F1 score. We fix per-switch memory to 100KB and vary the epoch length. As the epoch length increases, the F1 score of all tasks decreases. Although a shorter epoch goes with higher accuracy, the epoch length must be larger than the time for collecting tables plus identifying AFG, which typically takes 8-25ms (see Figure 12a).

Scalability of control-plane overhead (Figure 11a): We evaluate how the central-analyzer overhead scales with topology size by projecting the complexity model of § VI-B to fat-trees of k=4, 8, 16, 32 (20 to 1280 switches), using the per-table collection/analysis costs measured on our testbed (Figure 12a) as anchors. We make three observations. First, the per-epoch collection bandwidth grows linearly with the number of switches (from 3.3 Mbps at k=4 to 52 Mbps at k=16); this traffic is carried on the dedicated control network and, crucially, is *independent of the data-plane traffic volume*, since the analyzer processes aggregated per-epoch tables rather than per-packet records. Second, the notification bandwidth is negligible ($<$ 0.4 Mbps even at k=32) and the per-edge marking-table memory is a constant (a few hundred bytes for tens of AFGs). Third, the dominant cost is the table analysis: a naive *sequential* analyzer grows as $O(N_s)$ and would exceed a 5-second epoch beyond k=16, whereas the *layer-by-layer* collection of § VI-B processes each topology layer concurrently and keeps the critical-path latency bounded (about 50 ms, nearly constant in k). This confirms that, with layer-by-layer collection, AnoMon's control plane scales to large data-center topologies. We note that controller-free designs such as ISDC [82] and CFS [83] avoid a central analyzer altogether to remove the single-point collection bottleneck; however, they target distributed data collection and flow coverage rather than cross-switch per-hop anomaly localization. AnoMon deliberately retains a lightweight analyzer to perform the cross-switch correlation needed for network-wide localization, and the analysis above shows that this analyzer's overhead remains bounded and scalable.

Comparison with cooperative flow selection (Figure 11b): To position AnoMon against network-wide flow-selection systems, we reimplement CFS [83] and reproduce its coverage result: by making each flow the responsibility of a single on-path switch, CFS achieves near-optimal flow coverage, about 2.7$-$4.4$\times$ higher than independent Flow-Radar under the same counters. However, CFS targets covering *all* flows, so its counter budget must grow with the total number of flows (full coverage of 200K flows needs roughly 320K counters in our setting). AnoMon addresses a different goal: it does not attempt full flow coverage, but selectively monitors only the flows in AFGs. Hence, for anomaly localization, AnoMon's measurement budget grows with the number of abnormal flows rather than all flows, which is complementary to the coverage-oriented objective of CFS.

Impact of flow scale (Figure 11c): We vary the total number of flows from 25K to 200K (an 8$\times$ range) at a fixed per-switch memory and report the AFG detection F1. The F1 stays in a stable range (0.59 to 0.84) with no systematic degradation as the flow count grows, because AnoSketch identifies AFGs from the *group-level* aggregate signal, whose detectability depends on the abnormal intensity of a group rather than on the total number of flows in the network. This shows that AnoMon is not limited to a few hot flows: a flow group is flagged whenever its aggregate abnormal signal crosses the threshold, regardless of whether that signal comes from a few large flows or many small ones. We emphasize that the per-flow-group accuracy is ultimately bounded by the abnormal signal strength and the provisioned memory: the fine-grained localization inside the network requires the per-switch AnoTable memory to scale with the number of *abnormal* flows (not all flows), as quantified by the memory-accuracy tradeoff in Figure 10a. Finally, the /24 source prefix used in our experiments is only one representative grouping rule (corresponding to a subnet/rack); AnoSketch supports subset-sum queries over arbitrary flow sets, so operators may group flows by prefix, path, ingress-egress pair, tenant, or service, and the analyzer can choose the granularity that balances accuracy and resolution without re-configuring the data plane.

Benefit of the fast-trigger mode (Figure 13): We finally evaluate the optional fast-trigger mode (§ VI-A), where an edge device that observes a threshold-crossing sketch counter triggers AFG marking immediately instead of waiting for the



(a) Time for maintaining edge devices.

(b) Time for collecting & querying AnoTable.

(c) Time for querying quantiles of all flows.

(d) AnoSketch collection bandwidth.

(e) AnoTable collection bandwidth.

Figure 12: Performance of AnoMon on testbed.

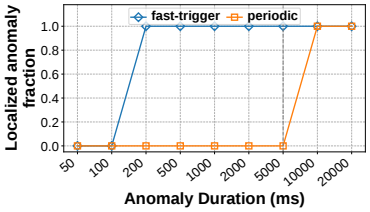


Figure 13: Benefit of the optional fast-trigger mode.

next periodic collection. Monte-Carlo sampling the onset of a first-time anomaly uniformly within an epoch, we report the fraction of anomalies whose per-hop localization is captured (*i.e.*, still active when the in-network measurement is activated). In the periodic mode, the in-network measurement is activated one epoch after detection, so per-hop distributions are captured only for anomalies lasting about two epochs ($\geq$10 s here). The fast-trigger mode instead activates it within $\sim$110 ms of the onset, capturing anomalies as short as $\sim$200 ms and cutting the mean detection latency from about half an epoch (2.5 s) to 0.13 s ($\sim$20$\times$), at the cost of a few additional control-plane messages.

C. Testbed Experiments

We use testbed experiments to evaluate the response time and monitoring overhead of AnoMon. As discussed in § VII, we implement AnoMon in a testbed built with 10 Edgecore Wedge 100BF-32X switches (with Tofino ASICs) and 8 end-hosts in the FatTree topology shown in Figure 2. All links run at 40Gbps. We use a client-server program to generate dynamic traffic [98] according to DCTCP [97] at each end-host, where the client generates requests through persistent TCP connections based on a Poisson process, and the server responds with the requested data. The source and destination of each flow are chosen uniformly at random. We periodically synchronize the clock of the central analyzer with the control plane of each switch using NTP [99]. When a new epoch starts, the central analyzer waits $10ms$ to ensure that all the packets from the previous epoch have exited (or been lost in) the network, and then starts to collect the sketches. We measure the time taken by system maintenance and problem analysis, and the bandwidth overhead of table collection.

Time of maintaining edge devices (Figure 12a): We find that AnoMon only takes $<25ms$ to maintain edge devices, which is significantly shorter than the epoch length. We measure the time for collecting AnoSketches, analyzing AnoSketches to identify AFG, and configuring edge switches to mark packets in AFG. We can see that the sketch collection time is always $<2.5ms$, and the configuration time is about 8-9ms. The analysis time grows as the per-switch memory increases. Under 224KB memory, the analysis time is 13.9ms.

Time of AnoTable collection and query (Figure 12b): We find that AnoMon takes $<200ms$ to collect the AnoTables and perform measurement tasks. We measure the time of collecting AnoTables and querying them to locate per-flow per-hop packet drops. The table collection times are always $<2.5ms$, which can be safely neglected. The analysis time grows as the per-switch memory increases, and when using 224KB memory, the analysis time is 190ms.

Time for acquiring fine-grained quantiles for all flows (Figure 12c): We find that AnoMon takes $<1.5s$ to acquire the per-flow distribution of all recorded flows. We measure the time for querying 8 quantiles (0.01, 0.05, 0.1, 0.5, 0.9, 0.95, 0.99) for all recorded flows. We can see that the query time grows as the per-switch memory increases, and when using 112KB memory, the analysis time is 0.62s.

Bandwidth of collecting AnoSketch (Figure 12d): We find that the extra bandwidth used for collecting all AnoSketches is $<$ 0.1Mbps. The extra bandwidth grows as the per-switch memory increases, and under 224KB memory, the extra bandwidth is 0.087Mbps, which is nearly negligible.

Bandwidth of collecting AnoTable (Figure 12e): The extra bandwidth used for collecting all AnoTables is $<$ 1.3Mbps, which grows as the per-switch memory increases. When using 224KB memory, the extra bandwidth is 1.28Mbps.

Summary: Overall, AnoMon achieves a small response time and lightweight monitoring overhead. When using 224KB per-switch memory, AnoMon only takes $<25ms$ to collect AnoSketches, identify AFG, and configure edge devices; and it takes $<200ms$ to collect AnoTables and locate network anomalies. The extra monitoring bandwidth of AnoMon is about 1.36Mbps, which is quite small compared to the link bandwidth (40Gbps). This is because AnoMon only monitors a small fraction of sampled large flows at the network edge, and a small fraction of flows that may experience incidents inside the network. AnoMon does not need to monitor all flows indiscriminately, and thus achieves reduced monitoring overhead.

Discussion and failure cases: We summarize the conditions under which AnoMon is most and least effective. Across the experiments, the accuracy improves with more edge/per-switch memory (which reduces sketch estimation variance and table collisions) and with a higher anomaly intensity (which strengthens the group-level signal), and it degrades when the abnormal signal is weak relative to the sketch noise. The always-on baselines (LightGuardian, Marple) lose accuracy under tight memory because they must amortize their memory over *all* flows, whereas AnoMon concentrates its fine-grained

budget on AFGs; NetSeer attains optimal packet-drop accuracy but, by reporting every event packet, incurs much higher bandwidth and does not generalize to distribution-based tasks. AnoMon is therefore most effective for group-correlated or recurring anomalies that produce a detectable aggregate signal. Its main failure modes are: (i) an isolated, short-lived, small-flow anomaly that never crosses the group-level threshold may be missed, for which AnoMon can be complemented by end-host triggers, finer flow groups, or a shorter epoch (at the cost of higher overhead); and (ii) a first-time anomaly shorter than one epoch may be detected only at the aggregate level without complete per-hop localization (§ VI-A). These boundaries reflect AnoMon's deliberate trade-off of instantaneous full-flow visibility for scalable, diagnosis-oriented measurement.

IX. Conclusion

This paper presents AnoMon, an accurate, low-overhead network measurement system that uses coarse-grained passive measurement at the network edge to guide fine-grained active measurement inside the network. Its two key components, AnoSketch and AnoTable, significantly outperform state-of-the-art solutions in measuring flow groups and estimating per-flow distribution, letting AnoMon perform various measurement tasks with high accuracy and low overhead. In future work, we intend to extend AnoMon to more tasks such as DDoS and SuperSpreader detection, and we hope its framework could inspire the design of future network monitoring systems.

Acknowledgments

This paper was supported in part by the National Natural Science Foundation of China (No. 62472418), the National Key Research and Development Program of China under Grant No. 2024YFB2906602, and the National Natural Science Foundation of China (NSFC) (No. 62372009).

References

[1] "Source codes of anomon," https://github.com/AnoMon2024/AnoMon 2024.
[2] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with switchpointer," in *Proc. USENIX NSDI*, 2018, pp. 453–456.
[3] V. Arun, M. Alizadeh, and H. Balakrishnan, "Starvation in end-to-end congestion control," in *Proc. ACM SIGCOMM*, 2022, pp. 177–192.
[4] Y. Zhou *et al.*, "Flow event telemetry on programmable data plane," in *Proc. ACM SIGCOMM*, 2020, pp. 76–89.
[5] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *Proc. USENIX OSDI*, 2016, pp. 233–248.
[6] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. USENIX NSDI*, 2013, pp. 29–42.
[7] E. Ghabashneh *et al.*, "A microscopic view of bursts, buffer contention, and loss in data centers," in *Proc. ACM IMC*, 2022, pp. 567–580.
[8] Y. Li *et al.*, "Hpcc: High precision congestion control," in *Proc. ACM SIGCOMM*, 2019, pp. 44–58.
[9] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.
[10] Y. Zhao *et al.*, "Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets," in *Proc. USENIX NSDI*, 2021, pp. 991–1010.
[11] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proc. ACM SIGCOMM*, 2017, pp. 85–98.
[12] W. Wang *et al.*, "Closed-loop network performance monitoring and diagnosis with spidermon," in *Proc. USENIX NSDI*, 2022, pp. 267–285.
[13] V. Sekar *et al.*, "csamp: A system for network-wide flow monitoring," 2008.
[14] B. Claise, "Cisco systems netflow services export version 9," Tech. Rep., 2004.
[15] M. Wang, B. Li, and Z. Li, "sflow: Towards resource-efficient and agile service federation in service overlay networks," in *24th Int. Conf. Distributed Computing Systems, 2004. Proceedings.*, 2004, pp. 628–635.
[16] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," in *Proc. ACM SIGCOMM*, 2015, pp. 479–491.
[17] J. Suh *et al.*, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *2014 IEEE 34th Int. Conf. Distributed Computing Systems*, 2014, pp. 228–237.
[18] D. Yu *et al.*, "dshark: A general, easy to program and scalable framework for analyzing in-network packet traces," in *Proc. USENIX NSDI*, 2019, pp. 207–220.
[19] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM*, 2018, pp. 561–575.
[20] Z. Liu *et al.*, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. ACM SIGCOMM*, 2016, pp. 101–114.
[21] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. ACM SIGMOD*, 2016, pp. 1449–1463.
[22] X. Chen *et al.*, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proc. ACM SIGCOMM*, 2020, pp. 226–239.
[23] C. Kim *et al.*, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, vol. 15, 2015.
[24] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM IMC*, 2010, pp. 267–280.
[25] T. Benson *et al.*, "Microte: Fine grained traffic engineering for data centers," in *Proc. ACM CoNEXT*, 2011, pp. 1–12.
[26] A. Dietmüller, G. Fragkouli, and L. Vanbever, "Poster: Learning distributions to detect anomalies using all the network traffic," in *Proc. ACM SIGCOMM*, 2023, pp. 1108–1110.
[27] A. Gupta *et al.*, "Sonata: Query-driven streaming network telemetry," in *Proc. 2018 conference of the ACM special interest group on data communication*, 2018, pp. 357–371.
[28] Y. Zhang *et al.*, "Cocosketch: high-performance sketch-based measurement over arbitrary partial key query," in *Proc. ACM SIGCOMM*, 2021, pp. 207–222.
[29] C. Masson, J. E. Rim, and H. K. Lee, "Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *arXiv preprint arXiv:1908.10693*, 2019.
[30] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," in *2016 ieee 57th annual symposium on foundations of computer science (focs)*, 2016, pp. 71–78.
[31] N. Ivkin *et al.*, "Streaming quantiles algorithms with small space and update time," *arXiv preprint arXiv:1907.00236*, 2019.
[32] ——, "Qpipe: Quantiles sketch fully in the data plane," in *Proc. 15th Int. Conf. Emerging Networking Experiments And Technologies*, 2019, pp. 285–291.
[33] "The NS.PY Discrete-Event Network Simulator," https://github.com/TL-System/ns.py.
[34] Y. Li *et al.*, "Flowradar: A better netflow for data centers," in *13th USENIX symposium on networked systems design and implementation*, 2016, pp. 311–324.
[35] ——, "Lossradar: Fast detection of lost packets in data center networks," in *Proc. 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016, pp. 481–495.
[36] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[37] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *Proc. Allerton Conf.*, 2011, pp. 792–799.
[38] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX NSDI*, 2014, pp. 71–85.
[39] Q. Huang *et al.*, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proc. ACM SIGCOMM*, 2020, pp. 404–421.
[40] R. Teixeira *et al.*, "Packetscope: Monitoring the packet lifecycle inside a switch," in *Proc. ACM SOSR*, 2020, pp. 76–82.
[41] Y. Lu *et al.*, "Counter braids: a novel counter architecture for per-flow measurement," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 1, pp. 121–132, 2008.
[42] S. Sheng, Q. Huang, and P. P. Lee, "Deltaint: Toward general in-band network telemetry with extremely low bandwidth overhead," in *Proc. IEEE ICNP*, 2021, pp. 1–11.
[43] A. Navaz, V. Sangeetha, and C. Prabhadevi, "Entropy based anomaly detection system to prevent ddos attacks in cloud," *arXiv preprint*

arXiv:1308.6745, 2013.
[44] S. Yu *et al.*, "Traceback of ddos attacks using entropy variations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 3, pp. 412–425, 2010.
[45] K. Kalkan, L. Altay, G. Gür, and F. Alagöz, "Jess: Joint entropy-based ddos defense scheme in sdn," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2358–2372, 2018.
[46] T. McBeath, "Method and apparatus for monitoring latency, jitter, packet throughput and packet loss ratio between two points on a network," Jun. 14 2011.
[47] X. Zhang *et al.*, "Ssl: A surrogate-based method for large-scale statistical latency measurement," *IEEE Trans. Serv. Comput.*, vol. 13, no. 5, pp. 958–968, 2017.
[48] Q. Yang *et al.*, "Deepqueuenet: towards scalable and generalized network performance estimation with packet-level visibility," in *Proc. ACM SIGCOMM*, 2022, pp. 441–457.
[49] F. Cangialosi, D. Levin, and N. Spring, "Ting: Measuring and exploiting latencies between all tor nodes," in *Proc. ACM IMC*, 2015, pp. 289–302.
[50] S.-P. Chan, C.-W. Kok, and A. K. Wong, "Multimedia streaming gateway with jitter detection," *IEEE Trans. Multimedia*, vol. 7, no. 3, pp. 585–592, 2005.
[51] I. Kartashevskiy and M. Buranova, "Calculation of packet jitter for correlated traffic," in *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, 2019, pp. 610–620.
[52] J. I. Munro and M. S. Paterson, "Selection and sorting with limited storage," *Theor. Comput. Sci.*, vol. 12, no. 3, pp. 315–323, 1980.
[53] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Approximate medians and other quantiles in one pass and with limited memory," *ACM SIGMOD Rec.*, vol. 27, no. 2, pp. 426–435, 1998.
[54] P. B. Gibbons, Y. Matias, and V. Poosala, "Fast incremental maintenance of approximate histograms," *ACM Transactions on Database Systems (TODS)*, vol. 27, no. 3, pp. 261–298, 2002.
[55] D. Felber and R. Ostrovsky, "A randomized online quantile summary in $o(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ words," *Theory Comput.*, vol. 13, no. 1, pp. 1–17, 2017.
[56] G. Cormode, A. Mishra, J. Ross, and P. Veselỳ, "Theory meets practice at the median: a worst case comparison of relative error quantile algorithms," in *Proc. ACM SIGKDD*, 2021, pp. 2722–2731.
[57] F. Zhao *et al.*, "Kll±approximate quantile sketches over dynamic datasets," *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1215–1227, 2021.
[58] J. He, J. Zhu, and Q. Huang, "Histsketch: A compact data structure for accurate per-key distribution monitoring," in *2023 IEEE 39th Int. Conf. Data Engineering (ICDE)*, 2023, pp. 2008–2021.
[59] J. Rasley *et al.*, "Planck: Millisecond-scale monitoring and control for commodity networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 407–418, 2014.
[60] Z. Liu *et al.*, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proc. ACM SIGCOMM*, 2019, pp. 334–350.
[61] B. Arzani *et al.*, "007: Democratically finding the cause of packet drops," in *Proc. USENIX NSDI*, 2018, pp. 419–435.
[62] A. Khandelwal, R. Agarwal, and I. Stoica, "Confluo: Distributed monitoring and diagnosis stack for high-speed networks," in *Proc. USENIX NSDI*, 2019, pp. 421–436.
[63] M. Moshref *et al.*, "Trumpet: Timely and precise triggers in data centers," in *Proc. ACM SIGCOMM*, 2016, pp. 129–143.
[64] A. Roy *et al.*, "Passive realtime datacenter fault detection and localization," in *Proc. USENIX NSDI*, 2017, pp. 595–612.
[65] Y. Wu, A. Chen, and L. T. X. Phan, "Zeno: Diagnosing performance problems with temporal provenance," in *Proc. USENIX NSDI*, 2019, pp. 395–420.
[66] C. Tan *et al.*, "Netbouncer: Active device and link failure localization in data center networks," in *Proc. USENIX NSDI*, 2019, pp. 599–614.
[67] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM SIGCOMM*, 2015, pp. 139–152.
[68] Y. Geng *et al.*, "Simon: A simple and scalable method for sensing, inference and measurement in data center networks," in *Proc. USENIX NSDI*, 2019, pp. 549–564.
[69] A. Adams, P. Lapukhov, and J. H. Zeng, "Netnorad: Troubleshooting networks via end-to-end probing," *Facebook White Paper*, 2016.
[70] F. Aubry, D. Lebrun, S. Vissicchio, M. T. Khong, Y. Deville, and O. Bonaventure, "Scmon: Leveraging segment routing to improve network monitoring," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.
[71] A. Dhamdhere *et al.*, "Inferring persistent interdomain congestion," in *Proc. ACM SIGCOMM*, 2018, pp. 1–15.
[72] R. Ben Basat *et al.*, "Pint: Probabilistic in-band network telemetry," in *Proc. ACM SIGCOMM*, 2020, pp. 662–680.
[73] G. Fioccola *et al.*, "Alternate-marking method for passive and hybrid performance monitoring," Tech. Rep., 2018.
[74] J. Sonchack *et al.*, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow," in *Proc. USENIX ATC*, 2018, pp. 823–835.
[75] C. Misa *et al.*, "Dynamic scheduling of approximate telemetry queries," in *Proc. USENIX NSDI*, 2022, pp. 701–717.
[76] Y. Lei *et al.*, "Printqueue: performance diagnosis via queue measurement in the data plane," in *Proc. ACM SIGCOMM*, 2022, pp. 516–529.
[77] E. C. Molero, S. Vissicchio, and L. Vanbever, "Fast in-network gray failure detection for isps," in *Proc. ACM SIGCOMM*, 2022, pp. 677–692.
[78] T. Holterbach *et al.*, "Blink: Fast connectivity recovery entirely in the data plane," in *Proc. USENIX NSDI*, 2019, pp. 161–176.
[79] X. Liu *et al.*, "Mozart: Temporal coordination of measurement," in *Proc. ACM SOSR*, 2016, pp. 1–12.
[80] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "FlowLens: Enabling efficient flow classification for ML-based network security applications," in *Proc. NDSS*, 2021.
[81] S. Kim *et al.*, "SketchFeature: High-quality per-flow feature extractor towards security-aware data plane," in *Proc. NDSS*, 2025.
[82] S. M. M. Mirnajafizadeh *et al.*, "Enhancing network attack detection with distributed and in-network data collection system," in *Proc. USENIX Security*, 2024.
[83] R. Ben Basat, G. Einziger, and B. Tayh, "Cooperative network-wide flow selection," in *Proc. IEEE/ACM IWQoS*, 2020.
[84] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *Proc. ACM SIGMOD*, 2018, pp. 1129–1140.
[85] "Supplementary materials of anomon," https://github.com/AnoMon2024/AnoMon2024/blob/main/AnoMon_Supplementary.pdf.
[86] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 3–14, 2014.
[87] T. M. Gil and M. Poletto, "Multops: A data-structure for bandwidth attack detection." in *Proc. USENIX Security*, 2001, pp. 23–38.
[88] C. Estan *et al.*, "Building a better netflow," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 245–256, 2004.
[89] M. Garetto and D. Towsley, "Modeling, simulation and measurements of queuing delay under long-tail internet traffic," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 47–57, 2003.
[90] Y. Geng *et al.*, "Exploiting a natural network effect for scalable, fine-grained clock synchronization," in *Proc. USENIX NSDI*, 2018, pp. 81–94.
[91] P. G. Kannan, R. Joshi, and M. C. Chan, "Precise time-synchronization in the data-plane using programmable switching asics," in *Proc. ACM SOSR*, 2019, pp. 8–20.
[92] "Barefoot tofino: World's fastest p4-programmable ethernet switch asics," https://barefootnetworks.com/products/brief-tofino/.
[93] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.
[94] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," *ACM SIGMOD Rec.*, vol. 30, no. 2, pp. 58–66, 2001.
[95] J. Guo *et al.*, "Sketchpolymer: Estimate per-item tail quantile using one sketch," in *Proc. ACM SIGKDD*, 2023, pp. 590–601.
[96] "The CAIDA UCSD Anonymized Internet Traces [2018]," https://www.caida.org/catalog/datasets/passive_dataset.
[97] M. Alizadeh *et al.*, "Data center tcp (dctcp)," in *Proc. ACM SIGCOMM*, 2010, pp. 63–74.
[98] W. Bai *et al.*, "Enabling ecn in multi-service multi-queue data centers," in *Proc. USENIX NSDI*, 2016, pp. 537–549.
[99] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.