LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets

Yikai Zhao[†], Kaicheng Yang[†], Zirui Liu[†], Tong Yang[†,§], Li Chen[¶],
Shiyi Liu[†], Naiqian Zheng[†], Ruixin Wang[†], Hanbo Wu[†], Yi Wang[‡,§], Nicholas Zhang[¶]

[†]*Department of Computer Science, Peking University, China*
[§]*Peng Cheng Laboratory, Shenzhen, China* [¶]*Huawei Theory Lab, China*
[‡]*Southern University of Science and Technology*

Abstract

[1] Network traffic measurement is central to successful network operations, especially for today's hyper-scale networks. Although existing works have made great contributions, they fail to achieve the following three criteria simultaneously: 1) **full-visibility**, which refers to the ability to acquire any desired per-hop flow-level information for *all* flows; 2) **low overhead** in terms of computation, memory, and bandwidth; and 3) **robustness**, meaning the system can survive partial network failures. We design LightGuardian to meet these three criteria. Our key innovation is a (small) constant-sized data structure, called *sketchlet*, which can be embedded in packet headers. Specifically, we design a novel SuMax sketch to accurately capture flow-level information. SuMax can be divided into sketchlets, which are carried in-band by passing packets to the end-hosts for aggregation, reconstruction, and analysis. We have fully implemented a LightGuardian prototype on a testbed with 10 programmable switches and 8 end-hosts in a FatTree topology, and conduct extensive experiments and evaluations. Experimental results show that LightGuardian can obtain per-flow per-hop flow-level information within $1.0 \sim 1.5$ seconds with consistently low overhead, using only 0.07% total bandwidth capacity of the network. We believe LightGuardian is the first system to collect per-flow per-hop information for all flows in the network with negligible overhead.

1 Introduction

Network traffic measurement is central to successful network operations, especially for today's hyper-scale networks with more than 10^5 devices [1–6]. Meanwhile, at end-hosts, knowing the traffic information in the core of the network can also benefit application performance [7–9]. To infer application performance and user experience, the community consensus is to measure at flow-level granularity. Thus, an ideal measurement system is expected to achieve: 1) **full-visibility**, which we define as the ability to acquire any desired per-hop flow-level information[2] for *all* flows. Typical desired information includes routing path, per-hop latency, jitters, and packet drops. 2) **lightweight** in terms of computation, memory, and bandwidth, independent of the scale of network and the traffic dynamics; 3) **robustness**: the system should survive partial network failures, including link failures, device failures, and bandwidth depletion [6, 10–12].

Although existing works have made great contributions, they fail to meet the above criteria simultaneously. We coarsely characterize them into four categories:

- **Partial/Sampling** solutions [13–17] only sample packets or flows, or collect detailed statistics based on a preconfigured list of conditionals [18, 19]. For instance, Everflow [20] samples each SYN packet, and Cisco switches use the "match" keyword to specify which network flows need to be counted. Therefore, only a subset of the network traffic is measured with questionable accuracy.
- **Probing** solutions [6, 21–24] measures the states of devices or links by sending probing packets, and only these probes are measured.
- **Sketch-based** solutions [25–34] collects the information of every packet in a compact data structure, namely sketch, on network devices. Current sketches are unable to collect important flow-level information, such as jitters and packet drops, and are not robust to network failures, particularly device failures. Most prior sketches cannot be implemented on P4-capable switches (§ 2.2).
- **In-band** solutions carry information in every packet header. AM-PM [35] cannot achieve full-visibility with only one bit per packet. Although INT [36, 37] can potentially achieve full-visibility, its bandwidth and processing overhead grows quickly with the scale of the network. In both the postcard [38] (mirroring packets on each switch) or the passport [36] mode (mirroring packets on only the sink switches), the number of packets is at least doubled, which is a huge burden for the network. *Flow [39] uses a cache to group packet-level telemetry information according to the flow IDs. But its bandwidth overhead is still proportional to the number of packets.

[1] Co-primary authors: Yikai Zhao, Kaicheng Yang, and Zirui Liu. Corresponding authors: Tong Yang (yangtongemail@gmail.com) and Yi Wang (wy@ieee.org).

[2] In this paper, per-hop flow-level information means per-flow per-hop information.

In summary, no existing work can achieve full-visibility without considerable performance overhead, and none focuses on lightweight and robust collection mechanism of network-wide flow-level information.

We design LightGuardian to meet the above three criteria simultaneously. Our key innovation is a (small) constant-sized data structure, called *sketchlet*. A sketchlet is a fragment of the sketch data structure on network devices (physical or virtual), carried in-band in a packet's header. At the end-host, LightGuardian collects the sketchlets, reconstructs the original sketch, and consequently obtains the accurate measurement results of all flows.

To support and make full use of sketchlets, LightGuardian incorporates three key techniques:

- **Accurate & versatile device-local sketches:** As current sketches fail to capture important flow-level statistics (per-hop latency and jitters), we design a novel sketch: the *SuMax sketch* to support common measurement tasks, as well as new tasks of operational importance. With our insight that recording both the sum and the maximum can accurately perform these tasks (§ 4.2), we design the sketch with two types of cells: the *sum cell*s and the *maximum cell*s. SuMax can be readily deployed on programmable network devices, and we have fully implemented it on a P4-capable switch (§ 7.1). Although SuMax is not the only way to measure flow-level statistics, it can support almost all measurement tasks thanks to its versatility.
- **In-band telemetry with sketchlets:** We propose a novel approach that combines in-band telemetry and device-local sketches. An INT-enabled device appends measurement data to each packet. INT alone consumes an enormous amount of bandwidth and multiplies the number of packets (§ 2.2). On the other hand, for sketch-based solutions, although sketch is a compact coding of flow-level information, the size of a sketch should be sufficiently large to ensure accuracy, thus cannot be embedded in packet headers. Combining the advantages of both approaches, our key novelty is to split the sketch with flow-level information into constant-sized sketchlets that can be embedded into selected packet headers. Since the number of flows in the network is much smaller than that of packets and the sketch is a compactly coded representation of flow-level statistics, the bandwidth overhead of sketchlets is significantly lower than that of INT, while accurate flow-level measurement can still be retrieved.
- **Incremental network-wide aggregation:** The receiving end-hosts can either forward the sketchlets to a global analyzer, or reconstruct the sketch locally to obtain measurement information of flows and devices inside the network. The information can assist end-host applications in performance optimization in a distributed fashion, which lessens the burden on the centralized network control/management plane. To guarantee robustness, we design the reconstruction algorithm to be tolerant of losses and reordering of sketchlets. Our algorithm can approximate a sketch with a subset of its sketchlets, and the reconstruction accuracy is incrementally improved with more arriving sketchlets. Our experimental results show that 80% sketchlets can achieve an accurate estimation (§ 8.1), while collecting 80% sketchlets only needs $1.0 \sim 1.5$ seconds.

To the best of our knowledge, LightGuardian is the first system to measure per-flow per-hop latency distribution and detect abnormal jitters with high accuracy for *all flows* on every participating network device, while maintaining low overhead. It also collects useful traffic data for operations and diagnostics previously unavailable in existing systems. Since LightGuardian aims to measure various per-hop flow-level information, after detecting end-to-end problems, users can use our system to locate culprit network devices. Besides, LightGuardian's on-device mechanism is not limited to physical devices, and can be readily used in cloud networking environments with virtualized network functions.

We have fully implemented a LightGuardian prototype on a testbed with 10 Tofino switches and 8 end-hosts in a FatTree topology. As a whole, our prototype can obtain per-flow per-hop information within $1.0 \sim 1.5$ seconds with consistently low overhead (0.07% of total bandwidth) on the network. We also conduct large-scale simulations using mininet [40] and P4 behavior model [41], confirming the correctness, robustness, and performance of LightGuardian. We release all source code anonymously [3].

In this paper, we make four key contributions:

We propose sketchlets and design a lightweight in-band telemetry system. Using sketchlets, our system makes the entirety of traffic information in the core of the network available at end-hosts for analytics and diagnostics. LightGuardian is lightweight, which takes up negligible bandwidth, and it can aggregate all sketchlets within 4 seconds.

We design the SuMax sketch to support common and more important measurement tasks with high accuracy. For common tasks (*e.g.*, flow size estimation), SuMax achieves 6.78 times smaller error rate. Further, LightGuardian can locate the culprit devices in the context of packet drops, inflated latency, and abnormal jitters, achieving almost 100% accuracy with less than 0.5MB memory.

We design an incremental reconstruction algorithm to achieve robustness and failure tolerance. Our experimental results show that, even when 50% end-hosts fail, the analyzer can still reconstruct 89% of all sketches. In addition, device and link failures do not affect the reconstruction of sketches of other devices (§ 8.3.2).

We implement a LightGuardian prototype and make it open-source. We also build a testbed and conduct extensive experiments, which confirms that our system can reach the design criteria.

[3] https://github.com/Light-Guardian/LightGuardian

Table 1: Comparison with the state-of-the-arts. In this table, "Impl." refers to implementation platforms; "P4 BM" refers to P4 behavior model [41]; "RMT" refers to Re-configurable Match Tables; and "PFPH" refers to "Per-Flow and Per-Hop".

Measurement Tasks		CM	FlowRadar	EverFlow	INT	AM-PM	LightGuardian
Device-local	Flow Size	✓	✓	×	✓	×	✓
	Flow Size Distr.	✓	✓	×	✓	×	✓
	Entropy	✓	✓	×	✓	×	✓
	Cardinality	✓	✓	×	✓	×	✓
Network-wide	PFPH Latency Distr.	×	×	×	✓	×	✓
	PFPH Packet Drops	×	×	×	✓	×	✓
	PFPH Jitters	×	×	×	✓	×	✓
	Forwarding Path	×	✓	✓	✓	×	✓
Impl.	RMT switches	✓	×	×	✓	✓	✓
	P4 BM	✓	✓	×	✓	✓	✓

2 Background and Related Work

2.1 Measurement Tasks

Existing measurement tasks can be classified into two categories: device-local measurement tasks and network-wide measurement tasks. Device-local measurement tasks refer to measuring flow-level information in a single node (an end-host, a switch or a router), and there have been various sketch-based solutions, such as sketches of CM [25], CU [26], Count [27], UnivMon [28], Elastic [29], SketchLearn [30], SketchVisor [42], and more [32–34, 43–48]. However, there are very few sketches designed for network-wide measurement tasks. This paper focuses on the following four network-wide measurement tasks.

1) Estimating Latency: We aim to estimate per-flow per-hop latency distribution. Existing works acquire end-to-end latency by sending probing packets. And they monitor specific flows by tracking their packets [49]. However, these solutions can hardly locate the victim flows and the culprit devices simultaneously. In contrast, per-flow per-hop latency distribution can help a lot but is more challenging.

2) Detecting Packet Drops: There are three causes of packet drops: random drops, loops, and blackholes. For random drops, the state-of-the-art LossRadar [50] uses a Bloom filter [51] and an Invertible Bloom Lookup Table (IBLT) [52] to accurately find drops. LossRadar works excellently in many cases. However, it consumes a lot of memory when a large flow drops many packets, which frequently happens when there are network misconfigurations [53]. For loops and blackholes, the state-of-the-art FlowRadar [54] also uses a Bloom filter and an IBLT, sharing the same advantages and shortcomings as that of LossRadar.

3) Detecting Abnormal Jitters: Jitters refer to drastic changes of packet inter-arrival time of a given flow. We aim to find abnormal jitters in the per-flow per-hop manner. Jitters are often caused by queuing, congestion, high bandwidth load, or network attacks. It can significantly affect the performance of streaming media (*e.g.*, audio, video, music). To detect jitters, end-to-end methods [55–57] have been proposed. However, they cannot work in the per-flow per-hop manner.

4) Tracing Forwarding Path: We aim to trace the forwarding path of any flow. Given a flow, tracing forwarding path can check whether the actual forwarding path is consistent with expectation. It can help test and/or debug new network protocols and network architectures, solutions for network congestion, load balance, and flow scheduling. Existing works for tracing forwarding path include FlowRadar [54], SwitchPointer [58], Service traceroute [59], and more [60–63].

2.2 Related Work

As shown in Table 1, compared with the state-of-the-art solutions, only our system supports device-local and network-wide measurement tasks, and it is implemented in both RMT switches (*e.g.*, Tofino) and P4 behavior model, achieving per-flow per-hop measurements. In this section, we mainly introduce the following four categories of measurement solutions. For other measurement solutions, please refer to references [64–72].

Partial/Sampling Solutions. Many measurement systems [14–17,20,46,73–76] are developed by sampling packets. Sampling can significantly reduce the overhead of both time and space, but inevitably sacrifices accuracy and misses important events. Typical systems include NetFlow [73], sFlow [74], OpenSketch [46], OpenSample [17], Everflow [20], NitroSketch [77], and more [14–16, 76]. The state-of-the-art UnivMon [28] obtains elegant theoretical guarantees using a multiple sample solution at the cost of high time complexities. Sampling solutions probably miss many small flows, and thus cannot achieve the ideal goal of fully-visibility.

Probing Solutions. These solutions monitor the network by sending tailored packets. Typical systems include Pingmesh [78], NetBoncer [6], NetSonar [22], NetNorad [23], and more [24, 79–82]. Recently, AM-PM [35] gains wide recognition in industry. AM-PM divides packet streams into time periods, and the middle packet in each period is essentially a probing packet. Therefore, it only records per-period packet information, but is unaware of flow-level information. Probing solutions cannot achieve the ideal goal of fully-visibility because they cannot measure per-flow information.

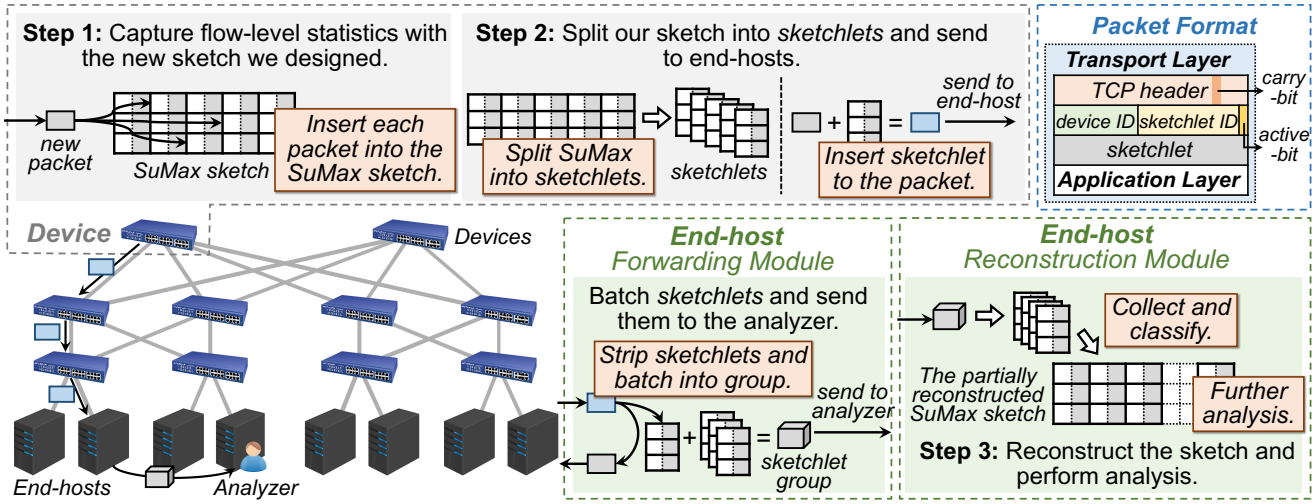


Figure 1: LightGuardian Overview and Workflow.

Sketch-based Solutions. There are a great number of sketch-based solutions, which can be further divided into three categories. First, design and optimization of sketch algorithms. Typical solutions include sketches of CM [25], CU [26], Count [27], Elastic [29], and more [28, 30–32, 34]. But they cannot measure the latency and jitters. Some are implemented in P4 behavior model, but only a very few (BeauCoup [33], Elastic [29]) are fully implemented in real programmable switches . Due to the limitations of RMT (Re-configurable Match Tables) based programmable switches, *e.g.*, limited concurrent memory access, single stage memory access, and *etc.*, the implementation in real switches is significantly more challenging. Thus, we aim to design a new sketch to support both device-local and network-wide measurement tasks, while can be easily implemented in RMT switches. Second, measurement systems with dedicated sketches. Typical solutions include FlowRadar [54], SketchLearn [30], NitroSketch [77], and more [33, 46, 50]. The dedicated sketches can barely achieve fully-visibility. Third, measurement systems with shining features. Typical solutions include Marple [83], Sonata [84], DREAM [85], Scream [86], and OmniMon [87]. By carefully designing the resource manager and telemetry operator, OmniMon [87] achieves both resource efficiency and high accuracy. Our SuMax sketch can also be applied in OmniMon. All existing solutions do not focus on the overhead with aggregating sketches all over the network.

In-band Telemetry Solutions. They insert packet-level information into every packet. Well-known solutions include INT [36] and its successor PINT [88]. INT is considered as the most promising solution for network measurement because of two reasons. First, it can achieve fully-visibility because it is flexible to carry any desired packet-level information. Second, it can be implemented in RMT switches in a per-packet manner. However, its bottleneck lies in the aggregation of INT information. The INT information is distributed in every network packet, and it is obviously very challenging to aggregate that per-packet information. INT has two aggregation strategies: postcard and passport, which mirror every packet with only INT information in each switch or only the sink switches. Although the INT information in each packet is small (*e.g.*, 100 bytes), the total bandwidth overhead is huge. What is worse, the number of packets in the network will be doubled, which is a heavy burden for packet processing. Another in-band telemetry solution *Flow [39] uses a cache to group packet-level telemetry information according to the flow IDs. In this way, some information (*e.g.*, 5-tuple flow ID) in one group is recorded only once. However, its bandwidth overhead is still proportional to the number of packets.

3 LightGuardian Overview

As shown in Figure 1, LightGuardian captures flow-level statistics on each participating network device (physical or virtual)[4] using sketches. The devices periodically split the sketches into sketchlets and send the sketchlets to the end-hosts by piggybacking them in headers of appropriate packets. Then at the receiving end, the end-hosts batch the sketchlets into groups and send them to a global analyzer when the network load is low. Finally, the analyzer reconstructs the sketches and perform analysis.

1) Capture flow-level statistics with novel sketches. LightGuardian captures flow-level statistics by deploying our SuMax sketch on each participating device. Every packet is processed into the sketch *without sampling*. Typical collected statistics include the flow size (number of packets/bytes), per-hop delay distribution, the arrival time of the last packet, and the maximum inter-arrival time. The above statistics are used for detecting packet drops and measuring per-hop latency and maximum inter-arrival time, which are essential tasks for industrial community. To support more tasks, we can also include more collected statistics, *e.g.*, the number of out-of-order packets, the highest sequence number, and *etc.*

2) Split sketches into sketchlets and send them to the end-hosts. The participating devices periodically split their

[4]Since most legacy switches and routers do not have programmable dataplane capabilities, they cannot participate in LightGuardian (and their existence in the network will not hinder the functions of LightGuardian).

sketches into sketchlets and send them to the end-hosts. Specifically, at the start of each measurement interval, the devices initiate a new SuMax sketch to record the flow-level statistics. In the end, the sketches are divided into sketchlets of several bytes (24 bytes in our implementation). Each sketchlet is one column[5] (or several columns) of the sketch. Each switch then attaches these sketchlets to appropriate incoming packets. Specifically, we choose the packets that have not yet carried a sketchlet with a fixed probability (*e.g.*, 0.05).

3) [Optional] Batch sketchlets and forward to a global analyzer. LightGuardian has two working modes: 1) Local analysis mode, where each end-host uses local sketchlets to perform analysis for local applications; 2) Global analysis mode, where a global analyzer collects all sketchlets and performs analysis for network operators. If an end-host does not want to perform local analysis, it can choose to forward sketchlets to a global analyzer. A system daemon process running on each end-host strips sketchlets off the packets, and maintains the received sketchlets. When the process has collected enough sketchlets, it batches the received sketchlets into groups and forward them to the global analyzer. For LightGuardian, more than 350 sketchlets are grouped into a UDP packet and share 42 bytes packet header, which significantly reduces the number of additional packets for measurement.

4) Reconstruct sketches and perform analysis. The end-host or the global analyzer (or end-hosts) can reconstruct the sketchlets into sketches and perform further analysis. The process of reconstructing sketches proceeds simultaneously with the process of collecting sketchlets. After collecting enough sketchlets, the analyzer can perform accurate estimation using the partially reconstructed sketch. According to our experiments, after receiving 55% sketchlets, our LightGuardian reports 90% valid results, while the average relative error (ARE) is only 0.088. Further, the estimation results are incrementally refined with more and more sketchlets collected, the ARE reduces to 1×10^{-2}, 1×10^{-3}, 2×10^{-4} when 80%, 90%, and 100% sketchlets are received, respectively.

In this way, LightGuardian well achieves the three mentioned design goals. For full-visibility, LightGuardian deploys SuMax sketch on each network device to monitor various per-flow per-hop information for all flows. For low overhead, LightGuardian uses small and constant-sized sketchlets to transmit measurement information, which makes the in-band overhead grow sub-linearly with the network/traffic scale. For robustness, the reconstruction process of LightGuardian does not require collecting all sketchlets whereas providing desirable accuracy. Besides, any end-host with limited computation resources can play the role of the global analyzer, which makes our system robust.

[5] A sketch consists of multiple bucket arrays, and a column refers to the buckets with the same index in each array.

4 Device-local Sketch Design: SuMax

4.1 Motivation

We design the SuMax sketch to achieve accurate measurement of flow-level information on network devices of different platforms: software (CPU, or OVS [89]), P4 behavior model [41], programmable switches. To make LightGuardian widely applicable, this paper focuses on P4 behavior model and programmable switch platforms, as the software implementation is straightforward. Using P4 also ensures our implementation can be compiled to available and future P4 back-ends, such as SmartNIC, FPGA and GPU.

UnivMon [28] and HashPipe [90] are implemented in P4 behavior model, but can hardly be implemented in RMT switches. To address these issues, Basat *et al.* proposed using a recirculate method [91], inevitably incurring complexities and degradation of switch throughput. BeauCoup [33] and Elastic [29] have been implemented in RMT switches (*i.e.*, Tofino switches) by complicated designs and programs. Further, the above four sketches cannot be directly used for network-wide measurement tasks, such as estimating latency and jitters. We found CM [25] is the most friendly sketch for programmable switches. On the one hand, we optimize its accuracy under the constraints of programmable switches. On the other hand, we extend its functions to support both device-local and network-wide tasks. In the meantime, we try to keep the designed sketches as simple as possible.

4.2 Rationale and Design Space for Sketches

We first introduce the well-known CM sketch [25]. It is a typical sketch algorithm that sums packet attributes (*e.g.*, packet number, bytes number). It uses d counter arrays $\mathcal{A}_0,\cdots,\mathcal{A}_{d-1}$. For each array, it has a hash function $\mathcal{H}_i(\cdot)$ to map a flow[6] uniformly and randomly into a counter. When a packet of flow f with attribute value α arrives, CM selects the counter $\mathcal{A}_i[\mathcal{H}_i(f)]$ for each array $\mathcal{A}_i$ and increments these counters by α. To query the attribute sum of flow f, CM returns the minimum value among $\mathcal{A}_0[\mathcal{H}_0(f)],\cdots,\mathcal{A}_{d-1}[\mathcal{H}_{d-1}(f)]$, which is still a sum of attributes of some flows. Therefore, CM has only over-estimation errors. Similarly, the CU sketch [26] increments only the smallest counter(s), significantly improving the accuracy but not supporting pipeline implementation.

We propose to record both of the sum value and the maximum value[7] to support versatile tasks. We insist that all packet attributes can be accurately estimated by keeping only the sum and maximum values. We also insist that either sum or maximum value is indispensable. For example, sketches of CM, CU, Count, FlowRadar cannot be used to find maximum latency or inter-arrival time and last arrival time, because they only record the sum value without maximum value.

[6] A flow has many packets sharing the same flow ID, which can be any combination of 5-tuple: source IP address, source port, destination IP address, destination port, protocol type.

[7] Note that [92] also suggests that the sketch algorithm can be used to find the maximum value in a sequence.

Table 2: Symbols frequently used in this paper.

Symbol	Meaning
f	An arbitrary flow
α	An attribute that needs to be recorded in the sum cell (*e.g.*, packet size)
β	An attribute that needs to be recorded in the maximum cell (*e.g.*, arrival time)
d	SuMax consists of d bucket arrays
w	Each array consists of w buckets
$\mathcal{A}_i$	The i-th bucket array
$\mathcal{A}_i^{sum}[\cdot]$	The *sum cell* in a bucket
$\mathcal{A}_i^{max}[\cdot]$	The *maximum cell* in a bucket
$\mathcal{H}_i$	A hash function from a flow to $\{0,\cdots,w\}$

4.3 Data Structure and Operations

Data Structure (Figure 2): Our SuMax consists of d bucket arrays $\mathcal{A}_0,\cdots,\mathcal{A}_{d-1}$. Each array $\mathcal{A}_i$ contains w buckets $\mathcal{A}_i[0],\cdots,\mathcal{A}_i[w-1]$. Each bucket has two cells: a *sum cell* and a *maximum cell*, recording the sum value and the maximum value of attributes, respectively. Each array $\mathcal{A}_i$ is associated with a hash function $\mathcal{H}_i(.)$ that maps a flow into one of its buckets. To support various tasks, we may need more than one sum value or maximum value in each bucket. For convenience, we only show using one sum value and one maximum value. Table 2 lists the frequently used symbols in this paper.

Insertion: To achieve high accuracy and support pipeline implementation, we propose an approximate conservative update strategy as follows. To record a packet of flow f with attribute α and β ($\langle f,\alpha,\beta\rangle$, α will be accumulated and β will be compared with the maximum), we first maintain a current minimum value ω and initialize it to ∞. For each array $\mathcal{A}_i$, we select a bucket $\mathcal{A}_i[\mathcal{H}_i(f)]$ by computing the hash function $\mathcal{H}_i(f)$. For each selected bucket $\mathcal{A}_i[\mathcal{H}_i(f)]$, we check its *sum cell* $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]$ and update it as follows:

- If $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]+\alpha<\omega$, update the current minimum value $\omega=\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]+\alpha$, and set the cell to ω.
- If $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]+\alpha\geqslant\omega$, and $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]<\omega$, set the cell to ω.
- If $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)]\geqslant\omega$, we keep the cell unchanged.

For the *maximum cell* $\mathcal{A}_i^{max}[\mathcal{H}_i(f)]$, we just set it to $\max\{\mathcal{A}_i^{max}[\mathcal{H}_i(f)],\beta\}$. The pseudo-code of the insertion operation is shown in Algorithm 1 in Appendix A.

Query: Given a flow f, SuMax returns two results: one sum value estimation and one maximum value estimation. The sum estimation is the minimum value among $\mathcal{A}_0^{sum}[\mathcal{H}_0(f)]$,$\cdots$, $\mathcal{A}_{d-1}^{sum}[\mathcal{H}_{d-1}(f)]$. The maximum value estimation is the minimum value among $\mathcal{A}_0^{max}[\mathcal{H}_0(f)]$,$\cdots$, $\mathcal{A}_{d-1}^{max}[\mathcal{H}_{d-1}(f)]$.

Example (Figure 2): To record a packet $\langle f,\alpha=3,\beta=4\rangle$, SuMax updates the d ($d=3$) buckets $\mathcal{A}_0[\mathcal{H}_0(f)]$, $\mathcal{A}_1[\mathcal{H}_1(f)]$, $\mathcal{A}_2[\mathcal{H}_2(f)]$ as follows. For the bucket $[6,3]$, we increase 6 to 9, set ω to 9, and set 4 to $\max\{4,3\}$. For the bucket $[9,7]$, as $9\geqslant\omega$ and $7\geqslant 4$, we keep this bucket unchanged. For the bucket $[3,5]$, as $3+\alpha<\omega$, we update ω to 6 and update 3 to

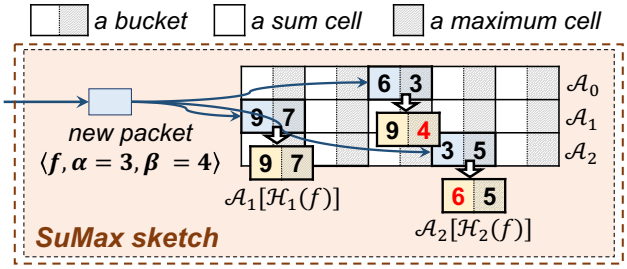


Figure 2: An example of SuMax.

$\omega=6$; as $4<5$, we do not change the maximum cell. After the insertion, when query flow f, SuMax returns $\min\{9,9,6\}=6$ as the sum estimation, and returns $\min\{4,7,5\}=4$ as the maximum value estimation.

Analysis: Our SuMax uses an approximate conservative update strategy to achieve both accuracy and pipeline friendly. Note that the conservative update strategy (CU) cannot be implemented in the pipeline because it needs the traceback operations to only increase the smallest counter(s). Our idea is to use the current minimum value to approximate the global minimum value. In each insertion process, with more and more counters accessed, the current minimum value will be closer and closer to the global minimum value, and thus the updated counter will be closer and closer to CU. Actually, the first array is updated following the rule of CM, and the last array is updated following the rule of CU. Since the counters in the last few arrays tend to have smaller values, they are more likely to be returned as query results. Therefore, SuMax can be viewed as an intermediate between CM and CU, and its error is also bounded between them, but closer to CU. As there are no tracebacks in our SuMax, it can be easily implemented in the switch pipeline.

4.4 Configuration of SuMax Sketch

In current implementation, we design each bucket as follows. Each bucket consists of four parts:

- a *flow-size cell* (sum cell) recording the flow size;
- λ_d *delay cells* (sum cells) recording the per-hop delay distribution, each one of which is associated with a predefined delay time interval;
- an *interval cell* (maximum cell) recording the maximum inter-arrival time;
- a *last-time cell* (maximum cell) recording the arrival time of the last packet of a flow.

All cells are initialized to zero. When the cells of a bucket are going to update, they should be updated as follows. Let t_{now} be the ingress timestamp of this packet, t_{last} be the value of the *last-time cell*, and $t_{interval}=t_{now}-t_{last}$. When $t_{last}=0$, we consider the current packet as the first packet of a flow, and set $t_{interval}=0$. First, we increment the *flow-size cell* by 1. Second, we select one cell from the λ_d *delay cells* according to the packet delay, and increment this cell by 1. Third, we compare the value in the *interval cell* with $t_{interval}$ and update it accordingly. Fourth, we update the *last-time cell* to t_{now}.

5 Transmission of Sketchlets

In this section, we show the transmission procedure of sketchlets in each participating network device. First, we split sketches into sketchlets. Second, we sample packets to carry sketchlets. Third, we select a sketchlet to be carried using our selection strategy and insert it into the packet.

5.1 Splitting the Sketch into Sketchlets.

LightGuardian deploys two SuMax sketches (one active and one idle) on each device. The active sketch is used to record flow-level information, while the idle sketch is split into sketchlets for transmission. After a fixed time interval (*e.g.*, 5s), we interchange these two sketches. We use an *active-bit* to indicate which sketch is active. The *active-bit* is flipped periodically, and the current interval is set to 5 seconds.

We split the idle sketch column by column, so that each sketchlet contains a column of buckets. Each sketchlet is associated with **1)** a *Sketchlet ID* indicating the column index; **2)** a *Device ID*; and **3)** the *active-bit* indicating which one of the two sketches it belongs to. The analyzer will sort the received sketchlets (bucket columns) according to the *Device ID*, *active-bit*, and *Sketchlet ID*.

5.2 Probabilistically Carrying Sketchlets.

Given an incoming packet, the device first checks the packet header: if it has already carried a sketchlet, no more sketchlets will be carried. Otherwise, the device calculates a fixed carrying probability λ_c (*e.g.*, 0.05) to determine whether this packet should carry a sketchlet. Each device samples only a part of the packets to carry sketchlets with λ_c, so that every device has a similar opportunity for packet transmission.

The packet format is shown in Figure 1. If a packet is selected to carry sketchlet, we insert the sketchlet between the TCP header and the application-layer message. First, we use a bit in the TCP header (*carry-bit*) to indicate whether this TCP packet carries a sketchlet. Second, we add a field to record the device ID (16 bits). Third, we add a field to record the sketchlet ID and the *active-bit*.

5.3 Sketchlets Selection: *K+chance Selection*.

Once the device determines the incoming packet should carry a sketchlet, we need an algorithm to choose a sketchlet. In-band telemetry solutions will lose measurement information when packet drops happen. To address this issue, we can send a sketchlet several times at the cost of more bandwidth usage. An effective solution is to use a counter array. Specifically, each counter corresponds to a sketchlet, indicating the number of times this sketchlet has been carried. For the incoming packet, we locate several counters by computing hash functions, find the smallest counter among them, and choose the corresponding sketchlet to carry. As mentioned above, similar to the CU sketch, this solution cannot be implemented in current P4-programmable switches, and thus we propose a new algorithm namely *k+chance selection*.

The *k+chance selection* uses k arrays, each of which is an N-bit array. For each array, each bit corresponds to a sketchlet. All bits are initialized to 0. Whenever we need to select one of the N sketchlets, we access the k arrays one by one. For each array, we randomly choose a bit: if it is zero, we choose the corresponding sketchlet and set this bit to 1; Otherwise, we access the next array. In the worst case, we do not find a zero bit after accessing all the k arrays, and we randomly choose one sketchlet to transmit. In this way, we only need to record an array ID in each sketchlet, which just takes $\lceil \log(k+1) \rceil$ bits (2$\sim$3 bits). By contrast, when using the simple round-robin, we need to record the column ID (usually 32 bits) in each sketchlet. *K+chance selection* is an approximately fair selection algorithm for hardware platforms. Our experiments show that *k+chance selection* works well (§ 8.1).

6 Reconstruction and Analysis

In this section, we first describe the two modules at the end-hosts: forwarding module and reconstruction module. These two modules can work in isolation or in parallel. Then we elaborate on how to obtain device-local measurements and network-wide analysis using SuMax.

6.1 End-host Modules

Reconstruction Module. This module dynamically classifies the received sketchlets into groups according to their device IDs and *active-bit*s, and sorts the sketchlets in each group by their sketchlet IDs. In this way, the end-host reconstructs a sketch for each group. Note that the reconstructed sketches might be incomplete because some sketchlets are still in the network or missing. Fortunately, an incomplete sketch can also be used to answer queries: each query will access d buckets, some of which may not have been received yet. We consider the values in these buckets as invalid, and report the minimum value among the other valid buckets. As long as one of the d buckets is valid, we can report a valid result. Otherwise, we report the result of invalid. In this way, after some sketchlets are collected, the end-host then uses these reconstructed sketches to perform further analysis. Our experimental results (see § 8.1) show that 55% sketchlets can report 90% valid results and achieve accurate estimation ($ARE < 0.1$). The following theorem provides theoretical guarantees for the reconstruction process.

Theorem 6.1 *After receiving sketchlets with a ratio of* θ*, SuMax can report valid results with a ratio of* $(1-(1-\theta)^d)$*. Specifically, when the result is valid, the estimated flow size has the following error bounds.*

$$\Pr\left\{|\hat{n}_f - n_f| > \varepsilon\right\} < \frac{\left(\frac{m\theta}{w\varepsilon} + 1 - \theta\right)^d - (1-\theta)^d}{1-(1-\theta)^d}$$

where d and w are parameters of SuMax (see Table 2), n_f and $\hat{n}_f$ are the real and estimated flow size, and m is the number of inserted packets.

The module can reconstruct the following four widely-studied [28, 29, 54] device-local measurements:

- *Flow Size Estimation.* We return the minimum value of the d mapped flow-size cells.
- *Flow Size Distribution.* We use the MRAC [93] algorithm with the first bucket array in SuMax as input.
- *Entropy.* We compute $-\sum(n_i \cdot \frac{i}{m} \log \frac{i}{m})$ based on the flow size distribution, where n_i is the number of flows with size of i, and $m = \sum(i \cdot n_i)$.
- *Cardinality.* We calculate the number of flows using the method of linear counting [94].

We believe the main advantage of this approach is that the measurement is done in a distributed fashion, without a centralized control or management plane.

Forwarding Module. End-hosts use this module to forward sketchlets to a global analyzer for network-wide analysis. Each end-host groups the received sketchlets into batches. The end-host will send a batch of sketchlets to the analyzer when appropriate. 1) When the bandwidth usage is high, the end-host does not send sketchlets. 2) When the number of the accumulated sketchlets reaches a threshold, or the end-host has not sent any sketchlets for a certain period, it will send all the accumulated sketchlets to the network-wide analyzer. The network-wide analyzer reconstructs the sketches as the end-hosts do, and then performs the network-wide analysis.

6.2 Network-wide Analysis with SuMax

For the following four network-wide analysis tasks, we need to access different SuMax cells for different tasks. To perform network-wide analysis tasks, we have two steps. First, the network operator detects abnormal end-to-end incidents (*e.g.*, TCP duplicate ACKs, TCP timeout[8]), and report the victim flows to the network-wide control plane analyzer. Second, based on the network topology, the analyzer further investigates the sketches on the switches in the forwarding path of the victim flow as to locate the specific culprit device or link.

Locating Inflated Latency. Locating inflated latency refers to finding out the culprit switch, and the victim flow when inflated end-to-end latency occurs. First, the end-host detects abnormal incidents of inflated end-to-end latency, and reports the ID of the victim flow. Second, the analyzer queries the per-hop latency distribution of this flow by accessing the delay cells in the corresponding reconstructed sketches. In this way, it can easily locate the culprit switches with inflated latency (*e.g.*, a switch on which 80% packets have $> 10\mu s$ latency).

Locating Packet Drops. As mentioned above, there are three main packet drops behaviors: random drops, loops, and blackholes. Random drops may result from hardware failures (*e.g.*, faulty interfaces in switches). Loops may result from the misconfiguration of the forwarding table, which leads the packets of the victim flows forever loop among several switches. Blackholes may result from forwarding entries corruption in culprit switches. After detecting end-to-end packet drops from TCP re-transmission, timeout, or ping probe loss, the end-host (sender) reports the flow ID to the analyzer. To locate the culprit switch, the analyzer queries the victim flow in every sketch on the forwarding path by accessing the flow-size cells. **1)** If the flow size suddenly drops to 0 after passing a switch, we report the switch as a blackhole. For example, suppose there are five switches ($s_1 \sim s_5$) on the forwarding path. If the estimated flow sizes on the five switches are $100, 100, 100, 0, 0$, respectively, we report s_4 as a blackhole. **2)** If the flow size is abnormally large on several switches, we infer a loop happens on them. For the same example with five switches, if the flow sizes are $100, 100, 5000, 5000, 0$, respectively, we infer that s_3 and s_4 probably be involved in a loop. **3)** If the flow size slightly decreases after passing a switch, we infer that the switch suffers random packet drops. For the same example, if the flow sizes are $100, 100, 95, 95, 95$, we infer random packet drops happen on s_3.

Locating Abnormal Jitters. After detecting end-to-end variation in the packet inter-arrival time of a flow, the end-host reports the flow ID to the analyzer. The analyzer queries the maximum inter-arrival time of that flow, and finds out the culprit switches on which the result is abnormally large.

Finding Abnormal Forwarding Path. When an end-host receives a packet which carries a sketchlet not belonging to the switches on the expected forwarding path, we report this packet suffers abnormal forwarding.

7 Prototype Implementation

In this section, we first describe the workflow and difficulties we face when implementing a LightGuardian prototype on a programmable switch (Tofino-40GbE). On each switch, we develop SuMax and the sketchlet transmission mechanism using P4 [96]. Then we overview the components in the end-hosts: the kernel modules to collect and forward the sketchlets.

7.1 SuMax on Programmable Switches

All existing sketches can be implemented in the software (*e.g.* middleboxes, virtual network appliances, *etc.*), but most of them cannot be deployed on programmable switches, which limits their applicability outside of cloud networking environments. For LightGuardian, since deployability is crucial to achieving full-visibility in all network environments, we first show that SuMax can be deployed on programmable switches by implementing it on a Tofino-40GbE switch.

7.1.1 Workflow

On the switch, we design the workflow (relevant to LightGuardian) (Figure 3) as follows: we put Decision Making Stage in the ingress pipeline, and Sketching Stage and Sketchlet Generation Stage are placed in the egress pipeline.

The **Decision Making Stage** decides the following:

[8] Some tools provided by the OS (*e.g.*, ePBF [95]) can help operators to easily detect these abnormal incidents.

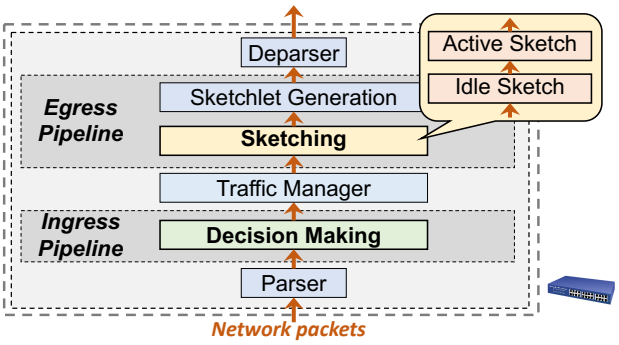


Figure 3: Workflow on an RMT switch.

- **Active sketch**, *i.e.* which sketch should be inserted into. As mentioned in § 5.1, we deploy two SuMax sketches on each switch, and use the *active-bit* to identify the *active* sketch. Note that the data plane of the switch *cannot* periodically update the active-bit. Therefore, we run a process in the switch control plane to periodically flip it. As the flipping is asynchronous, we forbid carrying sketchlets in the last second in each measurement interval.
- **Fitness for sketchlets**, *i.e.* whether the packet should carry a sketchlet. The fitness conditions are: 1) the packet is not carrying a sketchlet; 2) For each packet, we use its 5-tuple and its ingress timestamp to calculate a 16-bit hash value (CRC16), and only when the value falls within $[0, \lambda_c 2^{16})$, the packet is selected to carry a sketchlet. λ_c is a pre-configured parameter, and the second condition is approximately allowing a packet to carry sketchlet with a probability of λ_c
- **Sketchlets selection.** As described in § 5.3, we use the *k+chances selection* algorithm to select a sketchlet to carry. Thus, we need to randomly select a bit for each bit array. In Tofino switches, we can only achieve pseudo-randomness: we still use CRC16 to generate approximately random numbers, and choose reasonable polynomials of CRC16 to generate multiple approximately independent random numbers. Due to limitation of Tofino switch, we set $k = 1$.

In the **Sketching Stage**, we place two sketches: one idle and one active. Their status is periodically flipped. These two sketches are two match-action tables placed in the egress pipeline, so each packet will pass them sequentially. For each packet, the sketch table checks the active-bit. If the active-bit indicates the current sketch is active, we hash the flow ID to update the corresponding cells to record packet information. The update procedures of SuMax are challenging on Tofino, and we highlight the difficulties below (§ 7.1.2).

The **Sketch Generation Stage** reads the selected sketchlet and writes it into the metadata if the packet is selected.

7.1.2 Challenge of Sketching Stage

SuMax records multiple packet attributes (*e.g.*, flow size, delay distribution, last arrival time, maximum inter-arrival time). This requires multiple cells in each bucket. In Tofino switches, the cells in SuMax are stored in *registers*. A switch has 12 *Match-Action Units* (MAU), each of which contains up to two

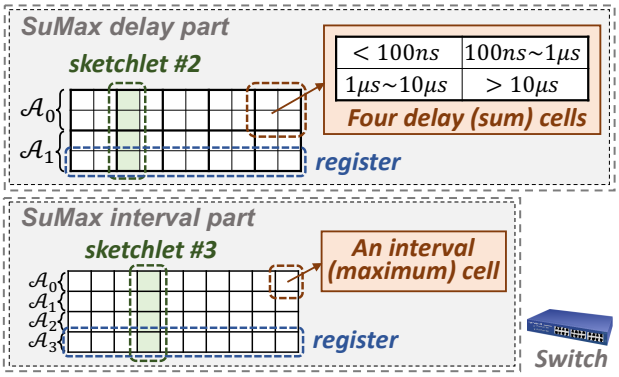


Figure 4: Sketch implementation on an RMT switch.

256KB registers. Since 6 MAUs are used in other stages, only 6 MAUs (12 registers) can be used in the Sketching stage. The main challenge is that, each incoming packet can only access each register exactly once, and each access can only read/write up to 64 consecutive bytes.

Thus, we have to assign the cells in a single bucket to multiple registers. In other words, we need to divide SuMax into parts. We use two examples, the measurement of latency distribution (sum) and that of maximum packet inter-arrival time, to illustrate our solution.

Latency distribution. We use the delay part of SuMax to perform this task. As shown in Figure 4, this part consists of $d = 2$ bucket arrays, each of which has $w = 2^{15}$ buckets. Each bucket has $\lambda_d = 4$ sum cells (32-bit), each of which corresponds to a predefined delay range. To make full use of the registers, we observe that:

- The four cells in each bucket should not be assigned to a single register. Since each 256KB register stores up to 2^{16} 32-bit cells, using a single register will limit the size of the sketch (up to 2^{14}), which compromises the accuracy.
- Using four registers to store the four cells in each bucket cannot be implemented on Tofino switch. As each switch has two sketches, each of which contains at least two bucket arrays, so we need at least 16 registers, while at most 12 registers are available in the Sketching stage.

Thus, we propose to use one register to store two cells in each bucket, as shown in Figure 4. We divide each bucket array into two registers, the first contains the first two cells of each bucket, and the second contains the remaining two cells. We group 4 cells in the same column into a sketchlet. In this way, either the active or the idle sketch is updated, each register is accessed *only once* for a packet.

Packet inter-arrival time is a task of measuring the maximum value, and its implementation is much easier. As shown in Figure 4, we set $d = 4$ and $w = 2^{16}$. For each bucket array, all 32-bit interval-cells are assigned to one register. We still group the 4 interval-cells in the same column into a sketchlet.

7.2 End-host Components

LightGuardian needs to implement three functions on the end-hosts: sending packets, receiving packets, reconstruction and performing analysis.

Sending packets: In the current implementation, LightGuardian inserts sketchlets between the Ethernet header and the IP header. However, we should emphasize that this is mainly due to hardware limitation: TCP checksum recalculation on Tofino is unreliable currently.

We also add the *carry-bit* after the Ethernet header, because there is no space in the Ethernet header. To implement this design, we program a Linux kernel module on the end-host, which registers a new packet type `ETH_P_SKETCHLET` in the Layer-3 protocol stack and modifies the Ethernet type of each packet to be sent to `ETH_P_SKETCHLET`, and allocates extra space for the *carry-bit*.

Receiving packets: We implement another Linux kernel module to handle `ETH_P_SKETCHLET` packets. This module decides whether the packet carries a sketchlet by checking the *carry-bit*, and records the sketchlet in the `stderr`.

Reconstruction and Forwarding We implement a forwarding module for end-hosts to forward the sketchlets to a centralized analyzer. It reads `stderr` every 1 millisecond. When the process finds the number of sketchlets in the log exceeds a threshold (dependant on Maximum Transmission Unit (MTU) of the network), or when a timeout is reached, the module generates a packet containing all the received sketchlets of the current interval, and sends it to the central analyzer. For example, when MTU is 9KB, the threshold is set to 350 packets ($\sim$8.4KB). We set the timeout to 100 milliseconds.

Finally, analysis can be performed on the end-host or the centralized analyzer with the same sketch reconstruction algorithm described in § 6.1.

8 Experimental Results

We conduct extensive experiments on a testbed and using mininet [40]. We focus on the following four key issues.

- **How accurate can our SuMax sketch measure per-flow statistics?** We implement our SuMax sketch using C++, and use the CAIDA datasets to evaluate the accuracy of SuMax for seven measurement tasks.
- **How much is the overhead of sending and aggregating sketchlets?** We generate network traffic following the widely used traffic distributions (WEB [97] and DCTCP [98]). We evaluate the aggregation time, the bandwidth overhead, and the impact on network performance (*e.g.*, RTT, FCT).
- **How accurate can LightGuardian detect network anomalies?** We use mininet to simulate a network, and evaluate the accuracy of LightGuardian in locating blackholes, loops, and abnormal jitters.
- **Is LightGuardian resilient to network failures?** We evaluate the performance of LightGuardian when end-hosts fail, or some sketchlets are missing.

We conduct the experiments using the following metrics: **ARE**, **RR**, **PR**, F_1 **Score**, **RE**, and **WMRE**. We explain the details of these metrics in Appendix C.

8.1 Experiments on SuMax

We use the anonymized IP traces collected in 2018 from CAIDA [99]. The dataset contains 6M packets belonging to 0.9M different flows. We set $d = 3$ by default, which means there are 3 bucket arrays in SuMax.

Flow size estimation (Figure 5a): We find that the accuracy of SuMax is higher than CM and close to CU. When using 96KB of memory, the ARE of SuMax is 6.78 times lower than CM, and 1.75 times higher than CU. We further study how the flow sizes affect the accuracy (see Figure 11a in Appendix D.1), and find that the results hold for both large and small flows.

Robustness (Figure 5b-5c): We find that partially reconstructed SuMax can provide accurate estimation. We set the memory to 768KB and measure the valid query rate and the ARE of the largest 1K flows. The results show that 55% reconstructed SuMax can report >90% valid results with <0.1 ARE, and 80% reconstructed SuMax can report >99% valid results with <0.01 ARE.

Other device-local tasks (Figure 5d-5e): We find that besides flow size estimation, SuMax also achieves good performance in other device-local measurement tasks, including estimating cardinality, flow size distribution (see Figure 11c in Appendix D.1), and entropy.

Delay distribution (Figure 5f): We find that the accuracy of SuMax is higher than CM and close to CU. We generate the delay of each packet according to the *chi-square distribution*. We set $\lambda_d = 8$ and vary w from 2^{10} to 2^{17}. For other delay distribution, please refer to Figure 12a-12e in Appendix D.1.

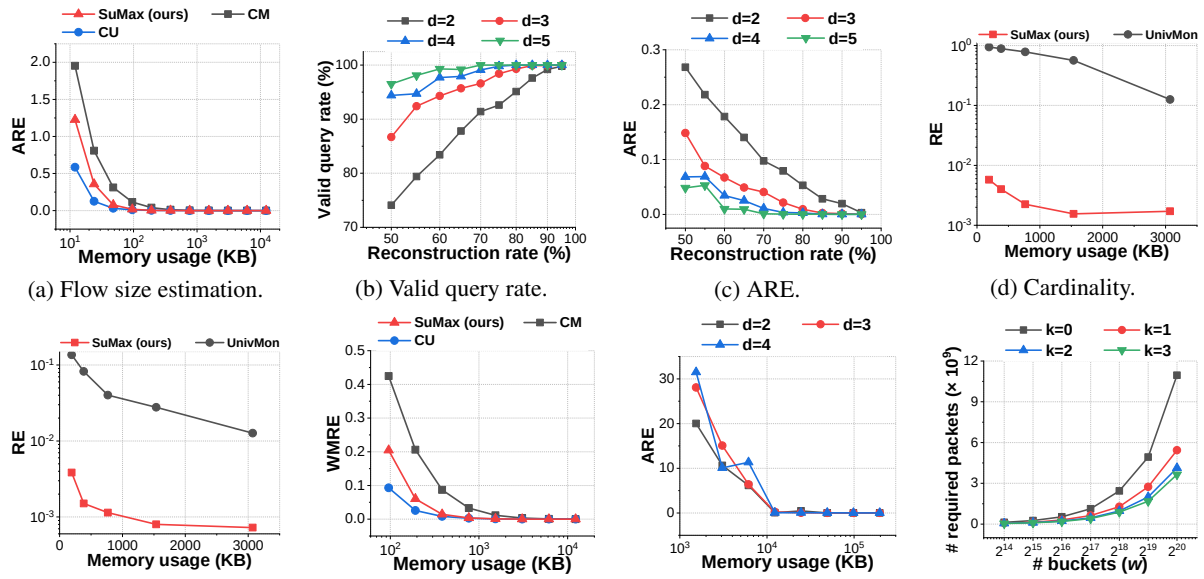
Maximum inter-arrival time (Figure 5g): We find that SuMax achieves <10 ARE when using more than 6MB of memory, and <0.3 ARE when using more than 12MB of memory. Since when abnormal incidents happen, the maximum inter-arrival time will rapidly increase dozens or hundreds times, <10 ARE is accurate enough to locate problems. We further study how the flow sizes affect the accuracy (see Figure 12h in Appendix D.1), and find that the results hold for both large and small flows.

***k+chance Selection* (Figure 5h):** We find that *k+chance Selection* can effectively reduce the number of packets required by the reconstruction process. According to the results, a larger k goes with fewer required packets, which demonstrates the effectiveness of our algorithm. We also find that the larger the w, the better the optimization effect.

We further study the memory overhead of SuMax, and find that its memory overhead grows sub-linearly with the network scale, which guarantees the scalability of LightGuardian (see Figure 11b and Table 3 in Appendix D.1).

8.2 Testbed Experiments

We evaluate LightGuardian on the testbed described in § 7. Take the delay distribution measurement task as an instance, the SuMax sketch we used contains $d = 2$ bucket arrays, each of which has $w = 2^{15}$ buckets, and each bucket contains 4



(a) Flow size estimation. (b) Valid query rate. (c) ARE. (d) Cardinality.

(e) Entropy. (f) Delay distribution. (g) Max inter-arrival time. (h) *K+chance Selection.*

Figure 5: Experimental results on SuMax and the selection algorithm.

delay cells. This sketch also supports locating packet drops and all the mentioned local measurement tasks. Similarly, we can use another SuMax to locate abnormal jitters. By default, we set the carrying probability λ_c to $\frac{1}{16}$, and set $k = 1$ for *k+chance selection*. We use two traffic distributions W1 (DCTCP [98]) and W2 (WEB [97]), which are widely used in existing works [8, 100–102]. On each switch, the number of the sketchlets is 2^{16}, and each sketchlet is 24 bytes.

Bandwidth overhead *v.s.* traffic load (Figure 6a): We find that our system saves substantial bandwidth than INT. We compare LightGuardian with a kind of INT that inserts 20-bytes per-packet information into the packet headers at each hop. The results show that the bandwidth overhead of LightGuardian ranges from 13.8Mbps to 25.7Mbps, which is only about 0.07% of the total bandwidth. The bandwidth overhead of INT ranges from 211Mbps to 394Mbps. Compared with INT, our LightGuardian saves more than 93.5% bandwidth. We also study how the carrying probability λ_c affects the bandwidth usage (see Figure 13a-13c in Appendix D.3).

FCT *v.s.* traffic load (Figure 6b): We find that LightGuardian has little impact on the network. We vary the bandwidth usage from 50% to 90%, and measure the average Flow Completion Time (FCT) before and after deploying LightGuardian. Under workload W1, after deploying LightGuardian, the average FCT increases by 8.3% at 50% traffic load, and 1.8% at 90% traffic load. Under workload W2, after deploying LightGuardian, the average FCT increases by 16.3% at 50% traffic load, and 5.6% at 90% traffic load. Even under 90% traffic load, LightGuardian still achieves <5ms FCT. We also study the impact of the flow size on the average FCT (see Figure 14b in Appendix D.3).

Per-hop latency *v.s.* traffic load (Figure 6c): We find that LightGuardian has little impact on the network. We test the per-hop latency before and after deploying LightGuardian in the network. We vary the bandwidth usage from 0% to 90%, and measure the average per-hop latency of 10^4 packets using the `ping -f` instruction. The results show that at 0% traffic load, after deploying LightGuardian, per-hop latency increases $1.6\mu s$. At 90% load, per-hop latency increases at most $3.1\mu s$.

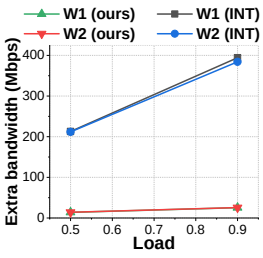
Reconstruction rate *v.s.* time (Figure 6d): We find that the sketches in LightGuardian can be quickly reconstructed. We use 90% of the total bandwidth and measure the reconstruction rate on each switch over time. The results show that under workload W1, the analyzer aggregates 90% sketchlets on the *edge switches*, the *aggregation switches*, and the *core switches* in 1.3, 1.7 and 2.1 seconds, respectively; and it aggregates 99% sketchlets in 2.1, 2.8 and 3.6 seconds, respectively. The results under workload W2 (as shown in Figure 14a in Appendix D.3) are similar. Other results related to sketch reconstruction are shown in Figure 14c-14d in Appendix D.3.

8.3 Simulations

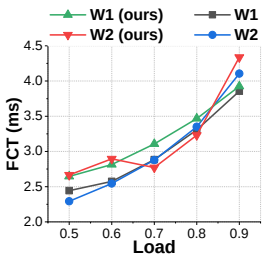
8.3.1 Simulations on Mininet

We evaluate LightGuardian's performance in locating blackholes, loops, and abnormal jitters through Mininet case studies. Our setup in Mininet consists of 16 hosts, 20 switches, and 48 links in a Fat-Tree topology. We only show the results as F1 scores. For more specific PR and RR results, please refer to Appendix D.2.

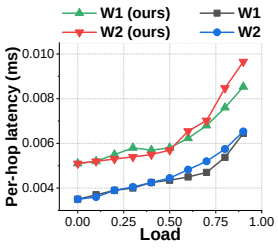
Locating blackholes (Figure 7a): We find that LightGuardian achieves high accuracy in locating blackholes. We randomly generate 10M packets belonging to 0.1M different flows. We create two blackholes by shutting down two links. And we reconstruct the sketchlets in a fixed time interval (5s) into sketches. For each flow, we query it in the reconstructed sketches to locate the culprit switches where the ratio $\frac{\mathcal{P}}{\mathcal{L}}$ is below a threshold. Here, for any switch, $\mathcal{P}$ is the estimated flow size, and $\mathcal{L}$ is the estimated flow size in the last-hop switch. The results show that when using 0.8MB of memory (2^{16} buckets), F1 score can reach 0.99.



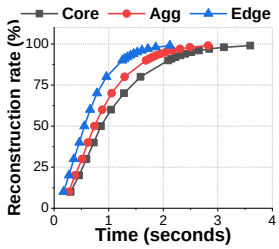
(a) Bandwidth *v.s.* load.



(b) FCT *v.s.* load.

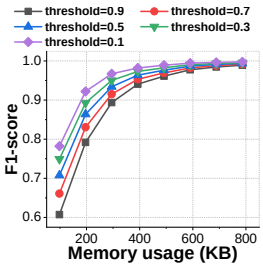


(c) Per-hop latency *v.s.* load.

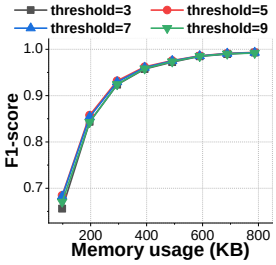


(d) Reconstruction *v.s.* time.

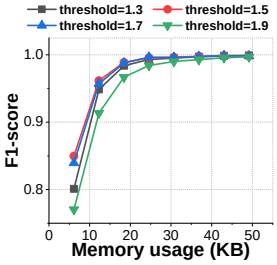
Figure 6: Experimental results on the testbed.



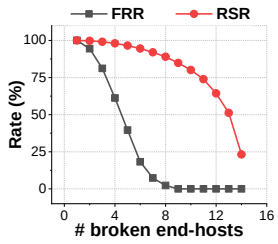
(a) Blackhole detection.



(b) Loop detection.



(c) Jitter detection.



(d) FRR/RSR *v.s.* λ_b.

Figure 7: Simulations results on Mininet.

Locating loops (Figure 7b): We find that LightGuardian achieves high accuracy in locating loops. We randomly generate 10M packets belonging to 0.1M different flows, and let 10% flows loop between two randomly selected adjacent switches. For each flow, we query it in the reconstructed sketches and locate the switches where $\frac{\mathcal{P}}{\mathcal{L}}$ exceeds a threshold. The results show that when using 0.8MB of memory, F1 score reaches about 0.99.

Locating abnormal jitters (Figure 7c): We find that LightGuardian achieves high accuracy in locating abnormal jitters. We randomly generate 10M packets belonging to 10K different flows. To simulate jitters on the switch, we randomly choose two links, and split each of them into two parallel links with different speed. In this way, the flows passing through the slow link will suffer jitters, which leads to a sharp increase in their inter-arrival time in the next-hop switch. The results show that when using more than 20KB of memory, the F1 score is close to 1.

8.3.2 Simulations for Robustness

Next, we focus on the robustness of LightGuardian. The network topology here is the same as Mininet. We set $w = 2^{16}$. And in each experiment, we randomly select λ_b end-hosts and shut them down[9]. Then we observe how many sketches can be fully-reconstructed (recovered) in the global analyzer. The metrics we used here are: 1) *Full-Recovery Rate (FRR)*: the probability of recovering all sketches; 2) *Recovering-Sketch Rate (RSR)*: the ratio of the number of recovered sketches to the number of all sketches; From Figure 7d, we find that our system is robust to survive several device failures. When $\lambda_b = 4$, the FRR is still >60%. Even if half of the end-hosts break down ($\lambda_b = 8$), the analyzer still stands a chance of recovering all sketches (FRR > 0). And the RSR slowly decreases as λ_b increases. When $\lambda_b = 7$, the analyzer can reconstruct more than 90% sketches.

[9]Normal end-hosts still send packets to broken end-hosts, but broken end-hosts cannot send packets to others.

9 Conclusion and Future Work

In this paper, we present LightGuardian, a full-visibility, lightweight, in-band network telemetry system. LightGuardian designs the SuMax sketch to capture per-flow per-hop statistics on the programmable data plane, and use the constant-sized sketchlet to aggregate the statistics to any end-host, which can then perform both the device-local and the network-wide analysis. Experiments on a testbed and mininet simulations show that our system is able to perform 4 local measurement tasks, 3 network-wide tasks, and 3 anomalies locating tasks with high accuracy and consistently low overhead.

In the future work, we plan to design a mechanism to automatically adjust the system parameters according to the current traffic characteristics; we plan to conduct large-scale simulations; we plan to design and evaluate other methods of transferring sketches; we plan to offload the reconstruction and forwarding modules in end-host to smart NIC; we plan to deploy our system in cloud networking; and we also plan to use our measurement results to further improve the performance of congestion control, load balancing, and traffic scheduling.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Z. Morley Mao, for their thoughtful suggestions. This work is supported by National Natural Science Foundation of China (NSFC) (No. U20A20179), and the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" (No. LZC0019).

References

[1] Minlan Yu. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review*, 49(1):11–17, 2019.

[2] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. Serverswitch: a programmable and high performance platform for data center networks. In *Nsdi*, volume 11, pages 2–2, 2011.

[3] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

[4] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: a new design element for low-latency dcns. *ACM SIGCOMM Computer Communication Review*, 44(4):283–294, 2014.

[5] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding network misconfigurations by automatic template inference. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 999–1013, 2020.

[6] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 599–614, 2019.

[7] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-rdma: Effective rdma middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM workshop on hot topics in networks*, pages 1–7, 2014.

[9] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 191–205, 2018.

[10] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.

[11] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.

[12] Qishi Wu, Sajjan Shiva, Sankardas Roy, Charles Ellis, and Vivek Datla. On modeling and simulation of game theory-based defense mechanisms against dos and ddos attacks. In *Proceedings of the 2010 spring simulation multiconference*, pages 1–8, 2010.

[13] JD Case, Mark Fedor, Martin Lee Schoffstall, and James Davin. Rfc1157: Simple network management protocol (snmp), 1990.

[14] Nick G Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM transactions on networking*, 9(3):280–292, 2001.

[15] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. Csamp: a system for network-wide flow monitoring. 2008.

[16] Vyas Sekar, Michael K Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 328–341, 2010.

[17] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 228–237. IEEE, 2014.

[18] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[19] Sajad Shirali-Shahreza and Yashar Ganjali. Flexam: flexible sampling extension for monitoring and security applications in openflow. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 167–168, 2013.

[20] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[21] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 139–152. ACM, 2015.

[22] Hongyi Zeng, Ratul Mahajan, Nick McKeown, George Varghese, Lihua Yuan, and Ming Zhang. Measuring and troubleshooting large operational multipath networks with gray box testing. *Mountain Safety Res., Seattle, WA, USA, Rep. MSR-TR-2015-55*, 2015.

[23] Petr Lapukhov and Aijay Adams. Netnorad: Troubleshooting networks via end-to-end probing. https://engineering.fb.com/networking-traffic/netnorad-troubleshooting-networks-via-end-to-end-probing/, Febrary 2016.

[24] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018.

[25] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.

[26] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3), 2003.

[27] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[28] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016.

[29] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM SIGCOMM*, pages 561–575. ACM, 2018.

[30] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590. ACM, 2018.

[31] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1449–1463, 2016.

[32] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.

[33] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 226–239, 2020.

[34] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1574–1584, 2020.

[35] Tal Mizrahi, Gidi Navon, Giuseppe Fioccola, Mauro Cociglio, Mach Chen, and Greg Mirsky. Am-pm: Efficient network telemetry using alternate marking. *IEEE Network*, 33(4):155–161, 2019.

[36] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. Inband network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[37] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. *ACM SIGCOMM Computer Communication Review*, 44(4):3–14, 2014.

[38] Cisco Nexus 9000 Series NX-OS Programmability Guide. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/92x/programmability/guide/b-cisco-nexus-9000-series-nx-os-programmability-guide-92x/b-cisco-nexus-9000-series-nx-os-programmability-guide-92x_chapter_0100001.html#id_95566.

[39] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 823–835, 2018.

[40] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIG-*

COMM Workshop on Hot Topics in Networks, pages 1–6, 2010.

[41] P4 behavior model. https://github.com/p4lang/behavioral-model.

[42] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126. ACM, 2017.

[43] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

[44] Qun Huang and Patrick PC Lee. Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1420–1428. IEEE, 2014.

[45] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: a novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):121–132, 2008.

[46] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, 2013.

[47] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. Sketching linear classifiers over data streams. In *Proceedings of the 2018 International Conference on Management of Data*, pages 757–772, 2018.

[48] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward nearly-zero-error sketching via compressive sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[49] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems*, pages 31–44. SIAM, 2020.

[50] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 481–495, 2016.

[51] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[52] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799. IEEE, 2011.

[53] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based performance diagnosis for network functions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 390–403, 2020.

[54] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, 2016.

[55] Leo Cloutier. Apparatus and method for correcting jitter in data packets, August 4 1998. US Patent 5,790,543.

[56] Siu-Ping Chan, C-W Kok, and Albert K Wong. Multimedia streaming gateway with jitter detection. *IEEE Transactions on Multimedia*, 7(3):585–592, 2005.

[57] Tom McBeath. Method and apparatus for monitoring latency, jitter, packet throughput and packet loss ratio between two points on a network, June 14 2011. US Patent 7,961,637.

[58] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 453–456, 2018.

[59] Ivan Morandi, Francesco Bronzino, Renata Teixeira, and Srikanth Sundaresan. Service traceroute: tracing paths of application flows. In *International Conference on Passive and Active Network Measurement*, pages 116–128. Springer, 2019.

[60] Feng Wang, Zhuoqing Morley Mao, Jia Wang, Lixin Gao, and Randy Bush. A measurement study on the impact of routing events on end-to-end internet path performance. *ACM SIGCOMM Computer Communication Review*, 36(4):375–386, 2006.

[61] Ying Zhang, Zhuoqing Morley Mao, and Jia Wang. A framework for measuring and predicting the impact of routing changes. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*, pages 339–347. IEEE, 2007.

[62] Kanak Agarwal, Eric Rozner, Colin Dixon, and John Carter. Sdn traceroute: Tracing sdn forwarding without changing network behavior. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 145–150, 2014.

[63] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *12th {USENIX} Symposium on Operat-*

ing Systems Design and Implementation ({OSDI} 16), pages 233–248, 2016.

[64] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM Computer Communication Review*, 44(4):407–418, 2014.

[65] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 71–85, 2014.

[66] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 421–436, 2019.

[67] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005.

[68] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 99–112, 2017.

[69] Omid Alipourfard, Masoud Moshref, and Minlan Yu. Re-evaluating measurement algorithms in software. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2015.

[70] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143, 2016.

[71] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. Mozart: Temporal coordination of measurement. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[72] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative network monitoring on a budget. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 467–482, 2018.

[73] Benoit Claise. Cisco systems netflow version 9. Technical report, 2004.

[74] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. Technical report, 2001.

[75] Pavlos Nikolopoulos, Christos Pappas, Katerina Argyraki, and Adrian Perrig. Retroactive packet sampling for traffic receipts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–39, 2019.

[76] Fangfan Li, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. A large-scale analysis of deployed traffic differentiation practices. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 130–144. 2019.

[77] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019.

[78] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45. ACM, 2015.

[79] Pietro Marchetta, Alessio Botta, Ethan Katz-Bassett, and Antonio Pescapé. Dissecting round trip time on the slow path with a single packet. In *International Conference on Passive and Active Network Measurement*, pages 88–97. Springer, 2014.

[80] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*, pages 327–340. USENIX Association, 2011.

[81] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 595–612, 2017.

[82] Srinivasan Seshan, Mark Stemm, and Randy H Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, pages 1–13, 1997.

[83] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[84] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special*

Interest Group on Data Communication, pages 357–371, 2018.

[85] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: dynamic resource allocation for software-defined measurement. *ACM SIGCOMM Computer Communication Review*, 44(4), 2015.

[86] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[87] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[88] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 662–680, 2020.

[89] The open virtual switch website. http://openvswitch.org.

[90] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. ACM, 2017.

[91] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018.

[92] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[93] Abhishek Kumar, Minho Sung, Jun Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):177–188, 2004.

[94] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[95] eBPF - Introduction, Tutorials Community Resources. https://ebpf.io.

[96] P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-checksums.

[97] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[98] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[99] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[100] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.

[101] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[102] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

APPENDIX

A Algorithm

Algorithm 1: Insertion of SuMax sketch

Input: A new packet $\langle f, \alpha, \beta \rangle$.

$\omega \leftarrow +\infty$;
for $i = 0 \rightarrow d-1$ **do**
  if $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)] + \alpha < \omega$ **then**
    $\omega \leftarrow \mathcal{A}_i^{sum}[\mathcal{H}_i(f)] + \alpha$;
    $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)] \leftarrow \omega$;
  else if $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)] < \omega$ **then**
    $\mathcal{A}_i^{sum}[\mathcal{H}_i(f)] \leftarrow \omega$;
  end
  $\mathcal{A}_i^{max}[\mathcal{H}_i(f)] \leftarrow \max\{\beta, \mathcal{A}_i^{max}[\mathcal{H}_i(f)]\}$;
end

B Proof of Theorem 6.1

Theorem B.1 *After receiving sketchlets with a ratio of θ, SuMax can report valid results with a ratio of $(1-(1-\theta)^d)$. Specifically, when the result is valid, the estimated flow size has the following error bounds.*

$$\Pr\left\{|\hat{n}_f - n_f| > \varepsilon\right\} < \frac{\left(\frac{m\theta}{w\varepsilon} + 1 - \theta\right)^d - (1-\theta)^d}{1-(1-\theta)^d},$$

where d and w are parameters of SuMax (see Table 2), n_f and $\hat{n}_f$ are the real and estimated flow size, and m is the number of inserted packets.

Proof B.1 *Let $\mathcal{A}_1[\mathcal{H}_1(f)], \cdots \mathcal{A}_d[\mathcal{H}_d(f)]$ be the values of d mapped buckets. Let $\mathcal{V}_1, \cdots, \mathcal{V}_d$ be d indicating random variables, where $\mathcal{V}_i$ indicates whether the i-th mapped bucket is received. Since the reported result of SuMax is valid as long as at least one buckets is received, the probability of acquiring a valid result is:*

$$\begin{aligned}\Pr\{\texttt{valid}\} &= \Pr\left\{\mathcal{V}_1 = 1 \vee \cdots \vee \mathcal{V}_d = 1\right\} \\ &= 1 - \prod_{i=1}^{d} \Pr\left\{\mathcal{V}_i = 0\right\} = 1 - (1-\theta)^d\end{aligned}$$

The expected number of packets mapped to each bucket is $\frac{m}{w} + n_f$. Since SuMax uses a conservative update method, not every packet increments the value of the bucket. Thus the value of each bucket satisfies:

$$\mathbb{E}\left(\mathcal{A}_i[\mathcal{H}_i(f)]\right) < \frac{m}{w} + n_f$$

According to the Markov inequality, we can derive that:

$$\Pr\left\{\left|\mathcal{A}_i[\mathcal{H}_i(f)] - n_f\right| > \varepsilon\right\} < \frac{\mathbb{E}\left(\mathcal{A}_i[\mathcal{H}_i(f)] - n_f\right)}{\varepsilon} < \frac{m}{w\varepsilon}$$

According to the total probability rule, we have

$$\begin{aligned}\Pr\left\{|\hat{n}_f - n_f| > \varepsilon\right\} &= \sum_{i=1}^{d} \Pr\{\zeta_i\} \cdot \Pr\left\{|\hat{n}_f - n_f| > \varepsilon \mid \zeta_i\right\} \\ < \sum_{i=1}^{d} \frac{\binom{d}{i}\theta^i(1-\theta)^{d-i}}{1-(1-\theta)^d} \cdot \left(\frac{m}{w\varepsilon}\right)^i &= \frac{\left(\frac{m\theta}{w\varepsilon} + 1 - \theta\right)^d - (1-\theta)^d}{1-(1-\theta)^d}\end{aligned}$$

where ζ_i indicates that there are i valid buckets after the reconstruction process.

C Evaluation Metrics

1) Average Relative Error (ARE): $\frac{1}{|\Psi|}\sum_{f_i \in \Psi} \frac{|n_i - \hat{n}_i|}{n_i}$, where n_i is the real statistics of flow f_i, $\hat{n}_i$ is the estimated statistics of flow f_i, and Ψ is the flow set.
2) Recall Rate (RR): The ratio of the number of correctly reported instances to the number of all correct instances.
3) Precision Rate (PR): The ratio of the number of correctly reported instances to the number of all reported instances.
4) F_1 **Score:** $\frac{2 \times PR \times RR}{PR + RR}$.
5) Relative Error (RE): $\frac{|Est. - True|}{True}$, where $Est.$ and $True$ are the estimated and true statistics, respectively.
6) Weighted Mean Relative Error (WMRE) [42, 93]: $\frac{\sum_{i=1}^{z} |n_i - \hat{n}_i|}{\sum_{i=1}^{z} \frac{n_i + \hat{n}_i}{2}}$, where n_i and $\hat{n}_i$ are the real and estimated event probabilities respectively, and z is the number of events.

D Additional Experimental Results

D.1 Experiments on SuMax

Flow size estimation (Figure 11a): We find that the ARE of SuMax is higher to CM and close to CU. When using 768KB of memory, for the largest 500 flows, the ARE of SuMax is 17.1 times lower than CM and only 0.1 times higher than CU.
Memory overhead (Figure 11b): We find that the memory overhead of SuMax grows sub-linearly with the number of generated packets, which guarantees the scalability of LightGuardian. We conduct this experiment in the flow size estimation task. We vary the number of generated packets in the network, and record how much memory SuMax needs to achieve 0.01 ARE. Table 3 further studies the memory overhead of SuMax in various tasks.
Delay distribution (Figure 12a-12e): We find that for different datasets, SuMax always achieves performance similar to CU. Figure 12a-12b show that the WMRE of SuMax is lower than CM and close to CU, which means SuMax has a stable performance. Figure 12c-12e show that when using 6MB of memory and varying the top-k flows, the WMRE of SuMax is similar to CU and lower than CM.
Last arrival time (Figure 12f-12g): We find that when using 768KB of memory, the Average Absolute Error (AAE) is less than 12ms; and when using 3MB of memory, the AAE is less than 1.5ms. Figure 12g further illustrated that when using

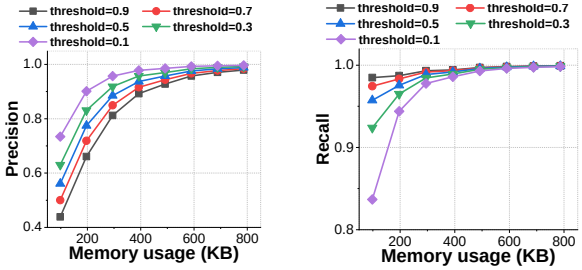
768KB of memory, the estimated results are absolutely correct for 87% packets.

Table 3: Memory usage of SuMax in various tasks.

Task	Target error	Memory (MB)
Flow size estimation	0.01	~ 0.1
Flow size distribution	0.05	~ 0.4
Cardinality	0.005	~ 0.2
Entropy	0.001	~ 0.8
Delay distribution	0.05	~ 0.8
Max inter-arrival	0.01	~ 50
Last arrival time	0.01	~ 0.8

D.2 Simulations on Mininet

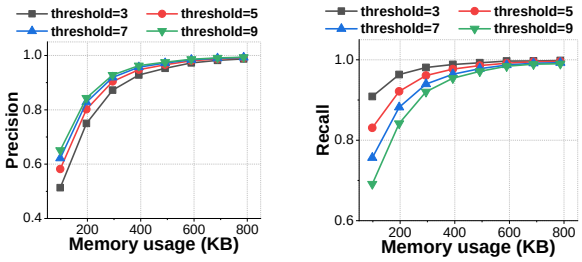
We demonstrate the specific PR and RR experimental results in § 8.3.



(a) PR *v.s.* memory usage. (b) RR *v.s.* memory usage.

Figure 8: Accuracy of locating blackholes.

Locating blackholes (Figure 8a-8b): We find that LightGuardian achieve high accuracy in locating blackholes. The results show that higher threshold goes with lower PR and higher RR. When using 800KB of memory (2^{16} buckets), the PR and RR reach $0.982 \sim 0.997$ and $0.998 \sim 0.999$ respectively.

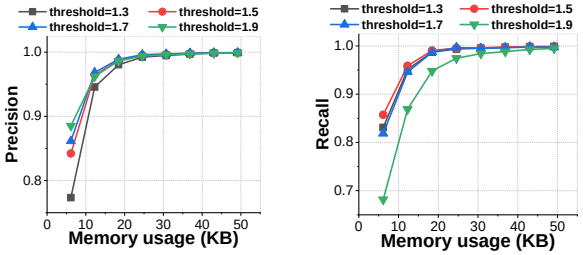


(a) PR *v.s.* memory usage. (b) RR *v.s.* memory usage.

Figure 9: Accuracy of locating loops.

Locating loops (Figure 9a-9b): We find that LightGuardian achieve high accuracy in locating loops. The results show that higher threshold goes with higher PR and lower RR. When using 800KB of memory, the PR and RR reaches $0.988 \sim 0.994$ and $0.993 \sim 0.998$ respectively.

Locating abnormal jitters (Figure 10a-10b): We find that LightGuardian achieve high accuracy in locating jitters. The results show that higher threshold goes with higher PR and lower RR. When using more than 500KB of memory, both PR and RR are close to 1.0. When using 50KB of memory (2^{12} buckets), the PR and RR reach $0.998 \sim 0.999$ and $0.994 \sim 0.999$ respectively.



(a) PR *v.s.* memory usage. (b) RR *v.s.* memory usage.

Figure 10: Accuracy of locating abnormal jitters.

D.3 Testbed Experiments

We further extend the experiments in § 8.2.

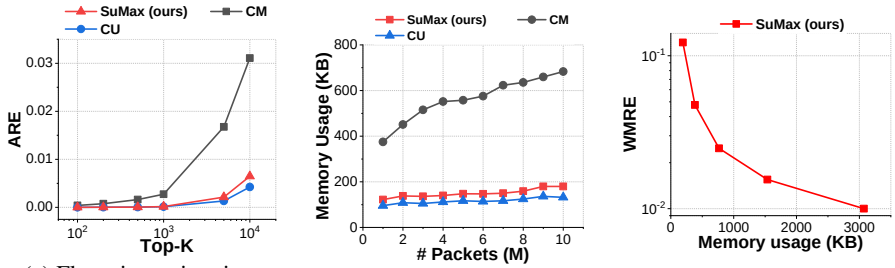
Bandwidth overhead *v.s.* λ_c (Figure 13a): We find that the bandwidth overhead of LightGuardian can be dynamically adjusted by the carrying probability λ_c. We vary the carrying probability λ_c from $\frac{1}{64}$ to $\frac{8}{64}$, and measure the bandwidth overhead. The results show that compared with INT, LightGuardian only uses 1.5% to 12.4% bandwidth. When the average packet size becomes smaller (*e.g.*, when encountering DDos attacks) and the number of the packets increases, our LightGuardian can adjust the bandwidth overhead by reducing λ_c. INT does not have this ability.

Required time (RT) *v.s.* λ_c (Figure 13b): We find that the time required to construct the sketches can be dynamically adjusted by the carrying probability λ_c. We generate 36Gbps traffic between two end-hosts in the same rack, and vary λ_c from $\frac{1}{64}$ to $\frac{8}{64}$. The results show that as λ_c increases, the required time decreases.

Required packets (RP) *v.s.* λ_c (Figure 13c): We find that the packets required to reconstruct the sketches can be dynamically adjusted by the carrying probability λ. The results show that the packets required to aggregate 90% and 99% sketchlets is negatively correlated to λ_c.

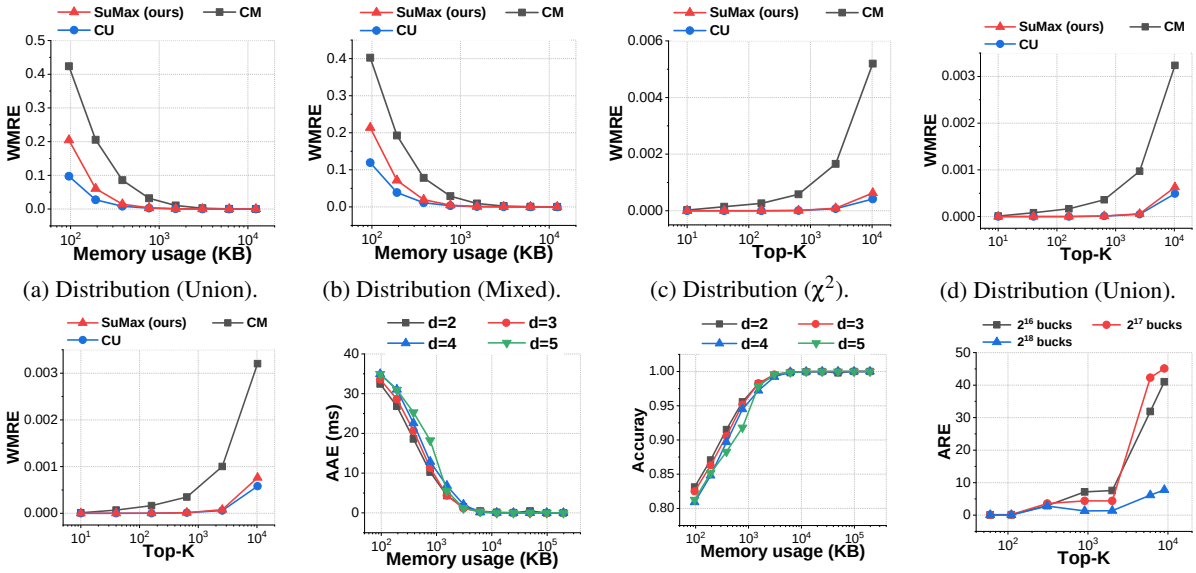
FCT *v.s.* flow size (Figure 14b): We find that LightGuardian has little impact on the FCT for the flows of any size. We measure the average FCT of flows of different sizes under 90% traffic load. We divide the flows into five groups according to their sizes: (0, 0.01MB), (0.01, 0.1MB), (0.1MB, 1MB), (1MB, 10MB) and (10MB, 100MB), and calculate the average FCT of each group. The results show that even for the flows of 100MB, the average FCT is no more than 40ms.

RP/RT *v.s.* w (Figure 14c-14d): We find that the required packets and the required time to reconstruct the sketches can be dynamically adjusted by w. We vary the number of sketchlets on the TOR switch from 2^{12} to 2^{16} and measure the number of the required packets and the required time to achieve certain reconstruction rates. The results show that both the required time and the required packets grow linearly with the number of sketchlets.



(a) Flow size estimation. (b) Memory overhead. (c) Flow size distribution.

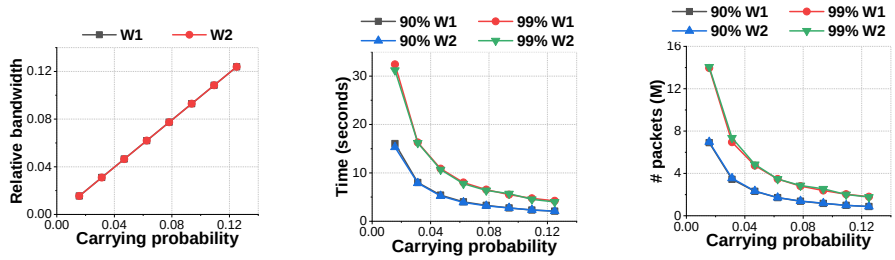
Figure 11: Experimental results of the SuMax sketch in device-local tasks.



(a) Distribution (Union). (b) Distribution (Mixed). (c) Distribution (χ^2). (d) Distribution (Union).

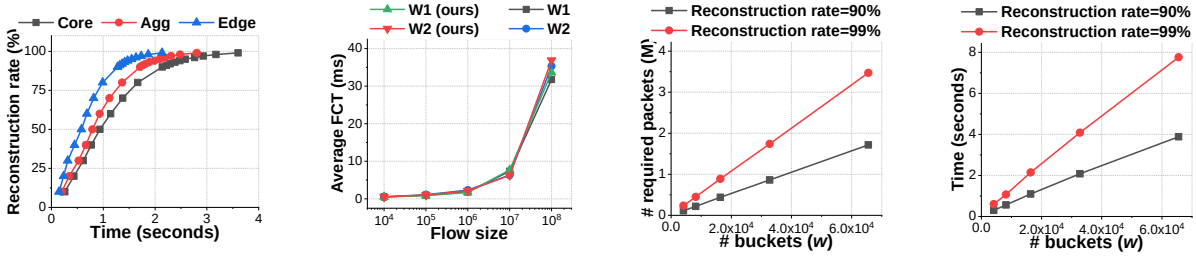
(e) Distribution (Mixed). (f) Last arrival time. (g) Last arrival time. (h) Max inter-arrival time.

Figure 12: Experimental results of the SuMax sketch in network-wide tasks.



(a) Bandwidth *v.s.* λ_c. (b) RT *v.s.* λ_c. (c) RP *v.s.* λ_c.

Figure 13: Impact of carrying probability λ_c.



(a) Reconstruction (W2). (b) FCT *v.s.* flow size. (c) RP *v.s.* w. (d) RT *v.s.* w.

Figure 14: Experimental results on the testbed.