# SketchVisor: Robust Network Measurement for Software Packet Processing

Qun Huang[1], Xin Jin[2], Patrick P. C. Lee[3], Runhui Li[1], Lu Tang[3], Yi-Chao Chen[1], Gong Zhang[1]
[1]Huawei Future Network Theory Lab [2]Johns Hopkins University [3]The Chinese University of Hong Kong

## ABSTRACT

Network measurement remains a missing piece in today's software packet processing platforms. Sketches provide a promising building block for filling this void by monitoring every packet with fixed-size memory and bounded errors. However, our analysis shows that existing sketch-based measurement solutions suffer from severe performance drops under high traffic load. Although sketches are efficiently designed, applying them in network measurement inevitably incurs heavy computational overhead.

We present SketchVisor, a robust network measurement framework for software packet processing. It augments sketch-based measurement in the data plane with a *fast path*, which is activated under high traffic load to provide high-performance local measurement with slight accuracy degradations. It further recovers accurate network-wide measurement results via compressive sensing. We have built a SketchVisor prototype on top of Open vSwitch. Extensive testbed experiments show that SketchVisor achieves high throughput and high accuracy for a wide range of network measurement tasks and microbenchmarks.

## CCS CONCEPTS

**•Networks → Network measurement;**

## KEYWORDS

Sketch; Network measurement; Software packet processing

**ACM Reference format:**
Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 14 pages.
DOI: http://dx.doi.org/10.1145/3098822.3098831

## 1 INTRODUCTION

Software packet processing is an important pillar of modern data center networks. It emphasizes programmability and extensibility, thereby supporting new network management features. Extensive work has been undertaken to improve software-based packet forwarding performance [16, 45]. Recent trends on network function virtualization (NFV) extend traditional layer 2-3 packet processing to more sophisticated middlebox functionalities via software-based control decisions [25]. Software switches, such as Open vSwitch [41], Microsoft Hyper-V Virtual Switch [32], and Cisco Nexus 1000V Virtual Switch [10], are now common building blocks of virtualization software and widely deployed in modern public and private clouds.

Network measurement is crucial to managing software packet processing platforms. Its goal is to collect essential network traffic statistics (e.g., heavy hitters, traffic anomalies, flow distribution, and traffic entropy) to help network operators make better network management decisions on traffic engineering, performance diagnosis, and attack prevention. Although network measurement has been well studied in IP networks, today's software switches, surprisingly, only support limited network measurement. For example, Open vSwitch only provides sampling-based measurement tools based on NetFlow [40] and sFlow [49], yet packet sampling inherently suffers from low measurement accuracy and achieves only coarse-grained measurement [28, 56]. While we can improve measurement accuracy by increasing the sampling rate or even recording all traffic (e.g., SPAN [51]), the resource usage will dramatically increase and pose scalability issues in high-speed networks.

*Sketches* provide an alternative to achieving fine-grained measurement. Unlike packet sampling, sketches are compact data structures that can summarize traffic statistics of *all* packets with fixed-size memory, while incurring only bounded errors. Many sketch-based solutions have been proposed in the literature to address different trade-offs between measurement accuracy and resource usage [13, 19, 22, 35, 46]. Although such proposals are not widely deployed in production IP networks due to the need of re-engineering switching ASIC, the programmability nature of software switches makes the deployment of sketch-based measurement in software packet processing viable. With the theoretical guarantees of resource usage of sketches, it is expected that sketch-based measurement incurs low overhead to the software packet processing pipeline.

Unfortunately, contrary to conventional wisdom, our analysis (§2.2) shows that existing representative sketch-based solutions in software actually consume substantial CPU resources, which could otherwise be used by other co-located applications (e.g., virtual machines or containers in virtualized environments). The root cause is that sketches are only primitives. While they are simple and efficient by design, applying them into practical network measurement requires additional extensions or components that often incur heavy computations. As modern data center networks now scale to 10Gbps or even higher speeds, sketch-based measurement will require excessive CPU resources to meet the line-rate requirement. Even though data centers do not always see high link utilization in practice [3], achieving line-rate measurement remains critical, especially in the face of traffic bursts, which indicate the presence


*SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA*

DOI: http://dx.doi.org/10.1145/3098822.3098831

of hot-spots or even attacks. Thus, not only do we require a measurement solution be resource-efficient under high traffic load, but also to accurately reason about the behavior of high traffic load.

We present SketchVisor, a robust network measurement framework for software packet processing. By robust, we mean that even under high traffic load, SketchVisor preserves both high (or even line-rate) performance and high accuracy for network-wide measurement. Instead of proposing a new sketch design, SketchVisor augments existing sketch-based solutions with a separate data path (called the *fast path*) that provides fast but slightly less accurate measurement for the packets that cannot be promptly handled by the underlying sketch-based solutions under high traffic load. Later, it accurately recovers network-wide measurement results from both sketch-based and fast path measurements.

Specifically, SketchVisor deploys a distributed data plane across software switches in the network, each of which processes packets based on the sketch-based measurement tasks as assigned by network operators, and redirects excessive packets to the fast path if the tasks are overloaded and cannot process those packets at high speed. We propose a new top-$k$ algorithm for the fast path to track large flows. By leveraging traffic skewness estimations and carefully designed data structures, our top-$k$ algorithm can achieve low amortized processing overhead and tight estimation bounds. We also maintain a global counter to track the traffic entering the fast path so as to capture the aggregate characteristics of small flows as well. Note that our fast path is general to support a variety of measurement tasks designed for different types of traffic statistics.

In addition, SketchVisor deploys a centralized control plane to merge the local measurement results (from both sketch-based and fast path measurements) from all software switches to provide accurate network-wide measurement. As the fast path inevitably loses information for high performance, we formulate a matrix interpolation problem to enable the control plane to recover missing information via compressive sensing [6, 7, 9, 61].

We have implemented a SketchVisor prototype and integrated it with Open vSwitch [41]. We have conducted extensive testbed experiments on SketchVisor for a wide range of measurement tasks and microbenchmarks. We show that for all our evaluated sketch-based measurement tasks, SketchVisor achieves above 17Gbps throughput with a single CPU core and near-optimal accuracy with only few KBs of memory in the fast path.

# 2 BACKGROUND AND MOTIVATION

We introduce the network measurement tasks considered in this paper, and demonstrate the overhead of existing sketch-based solutions in software.

## 2.1 Network Measurement

We target general measurement tasks that monitor traffic and collect traffic statistics, conducted by network operators, over one or multiple time periods called *epochs*. Traffic statistics can be either flow-based (identified by 5-tuples) or host-based (identified by IP addresses); or either volume-based (measured by byte counts) or connectivity-based (measured by distinct flow/host counts). This paper focuses on the following common traffic statistics that have been extensively studied in the literature.

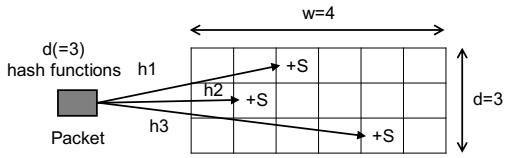



**Figure 1: Example of Count-Min sketch.**

- **Heavy hitter:** a flow whose byte count exceeds a threshold in an epoch.
- **Heavy changer:** a flow whose change of byte counts across two consecutive epochs exceeds a threshold.
- **DDoS:** a destination host that receives data from more than a threshold number of source hosts in an epoch.
- **Superspreader:** a source host that sends data to more than a threshold number of destination hosts in an epoch (i.e., a superspreader is the opposite of a DDoS).
- **Cardinality:** the number of distinct flows in an epoch.
- **Flow size distribution:** the fractions of flows for different ranges of byte counts in an epoch.
- **Entropy:** the entropy of flow size distribution in an epoch.

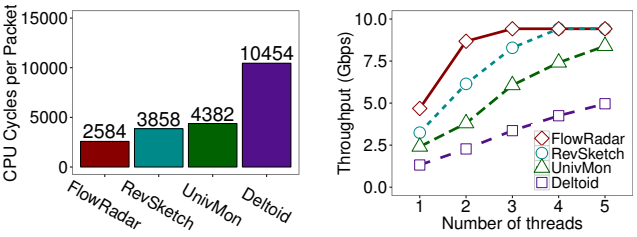
## 2.2 Performance Analysis

This paper focuses on sketch-based measurement, which summarizes traffic statistics of all observed packets with theoretical guarantees on memory usage and error bounds. At a high level, a *sketch* is a compact data structure comprising a set of *buckets*, each of which is associated with one or multiple *counters*. It maps each packet to a subset of buckets with independent hash functions, and updates the counters of those buckets. Network operators can query the counter values to recover traffic statistics.

The actual sketch design varies across measurement tasks. To show the main idea of sketches, we use a Count-Min sketch [14] as an example to illustrate how it collects flow-based traffic statistics. As shown in Figure 1, a Count-Min sketch consists of a two-dimensional array with $w$ columns and $d$ rows. For each packet, we hash its flow ID (5-tuple) to a bucket in each of the $d$ rows using $d$ independent hash functions, and then add the packet size to the counter of each bucket. To recover the size of a given flow, we use the minimum of the counters of the $d$ hashed buckets as an estimate. With proper settings of $w$ and $d$, the estimation flow size provably incurs a bounded error with a high probability [14].

This example shows that sketches perform fairly simple operations, mainly hash computations and counter updates. Intuitively, they should add limited overhead to software packet processing. Unfortunately, we find that this intuition does not hold in practice.

**Observations:** Sketches are only primitives that cannot be directly used for network measurement; instead, we must supplement them with additional components and operations to fully support a measurement task. In particular, in order to collect meaningful traffic statistics, we must add extensions to sketches to make them *reversible*, meaning that sketches not only store traffic statistics, but also efficiently answer queries on the statistics. For example, a Count-Min sketch can return a flow size only if we query a specific flow. Thus, if we want to identify, say, heavy hitters that exceed a pre-specified threshold, a Count-Min sketch can immediately report




(a) CPU cycles (b) Throughput with 10 Gbps NIC

**Figure 2: CPU overhead and throughput of sketch-based solutions.**

a heavy hitter after updating a packet by checking if the estimated flow size exceeds the thresholds based on the counters of the hashed buckets. However, the prior threshold is often unavailable in advance in practice, and we need to query for heavy hitters subject to different thresholds. In this case, we must query all candidate flows in the entire flow space and check if each of them exceeds a threshold. The flow space size can be extremely large, say $2^{104}$ for 5-tuple flows, thereby leading to substantial query costs.

Some proposals extend sketches with efficient reversibility. Deltoid [13] encodes flow headers with extra counters in each bucket and updates these counters on every packet. Reversible Sketch [46] partitions a flow header and hashes each sub-header into smaller subspaces. FlowRadar [28] maps flows to counters through XOR operations, such that new flows can be reconstructed by repeatedly XOR-ing the counters with known flows. However, such extensions incur heavy computational overhead.

In addition, most sketch-based solutions are designed for specific measurement tasks and traffic statistics. To run multiple measurement tasks together, we need to deploy each corresponding solution separately. Thus, running all of them on every packet becomes computationally burdensome. The recently proposed UnivMon [30] allows a single sketch to simultaneously collect different types of traffic statistics. However, it needs to update various components (including CountSketch [8] and top-$k$ flow keys), and remains computationally expensive (see analysis below).

**Microbenchmark:** To validate the above claims, we present microbenchmark results on the software implementations of four representative sketch-based solutions, namely Deltoid [13], Reversible Sketch [46], FlowRadar [28], and UnivMon [30], on heavy hitter detection. Here, we only measure the overhead of recording packets into each sketch-based solution but not collecting the recorded traffic statistics, as the latter can usually be done offline. We employ the same configurations as detailed in §7. Figure 2(a) shows the number of CPU cycles (measured by `Perf` [42]) of recording a packet in each solution. FlowRadar is the fastest and spends 2,584 cycles per packet, while Deltoid is the slowest and spends 10,454 cycles per packet. Such high CPU overhead translates to low throughput under high traffic load. Figure 2(b) shows the maximum throughput achievable by the four solutions versus the number of threads. No solution can achieve over 5Gbps with one thread; and Deltoid barely achieves 5Gbps even with five threads. Thus, these solutions, while being fast enough under low traffic load, become computationally intensive and resource demanding under high traffic load in modern data centers, in which servers are now commonly equipped with 10Gbps NICs and above.

We further analyze the breakdown of the CPU cycles in each sketch-based solution, and find that the performance bottlenecks vary across sketch-based solutions. For example, FlowRadar and Reversible Sketch incur more than 67% and 95% of CPU cycles, respectively, on hash computations (including randomizing flow headers to resolve hash collisions). Deltoid's main bottleneck is on updating its extra counters to encode flow headers, and this accounts for more than 86% of CPU cycles. UnivMon spends 53% and 47% of CPU cycles on hash computations and heap maintenance, respectively. The variations of performance bottlenecks also imply that optimizing specific functions (e.g., using hardware-based hash computations) may not work well for all sketch-based solutions.

Recent work [1] advocates that simple hash tables would suffice for network measurement due to improved cache management in servers and skewness of real-life traffic patterns. Although hash tables incur fewer computations than sketches [1], they consume significant memory usage (§7.6). Some systems [21, 29, 38, 62] attempt to filter traffic by predefined rules, so as to reduce memory usage. However, it requires manual efforts to configure proper rules to achieve both high accuracy and memory efficiency simultaneously. On the other hand, sketches provide theoretical guarantees on memory usage and error bounds, yet incur high computational overhead. Although they have not yet been widely deployed, we believe that their sound theoretical properties make them a promising building block for network measurement. Our work is to mitigate the computational overhead of sketch-based measurement, while preserving the theoretical guarantees of sketches.

## 3 SKETCHVISOR OVERVIEW

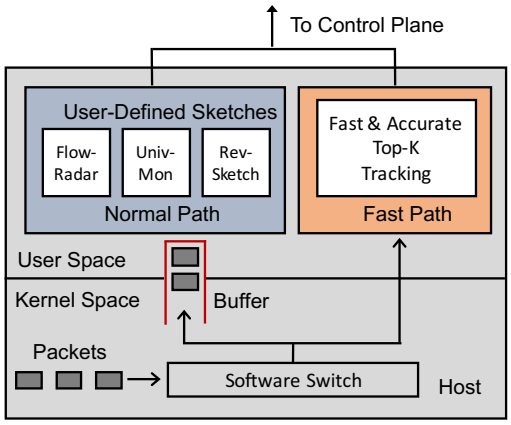
SketchVisor is a robust network measurement framework for software packet processing, with several design goals:

- **Performance:** It processes packets at high speed and aims to fulfill the line-rate requirement of the underlying packet processing pipeline.
- **Resource efficiency:** It efficiently utilizes CPU for packet processing and memory for data structures.
- **Accuracy:** It preserves high measurement accuracy of sketches.
- **Generality:** It supports a wide range of sketch-based measurement tasks.
- **Simplicity:** It automatically mitigates the processing burdens of sketch-based measurement tasks under high traffic load, without requiring manual per-host configurations and result aggregations by network operators.

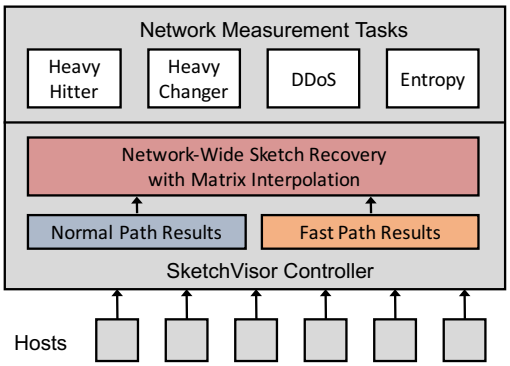
SketchVisor's design follows the line of *software-defined measurement* [23, 30, 36, 37, 56]. It comprises a distributed data plane that runs on the software switches of multiple hosts in a network, and a centralized control plane that aggregates the local results of all software switches and returns network-wide measurement results. Figure 3 shows both data-plane and control-plane architectures of SketchVisor.

### 3.1 Data Plane

The data plane (Figure 3(a)) deploys a measurement module in the software switch of each host. Each module processes incoming continuous packet streams and collects traffic statistics for the host. To avoid duplicate measurement, we can choose to monitor only




(a) SketchVisor data plane.




(b) SketchVisor control plane.

**Figure 3: SketchVisor architecture.**

either ingress or egress traffic, or leverage hash-based selection to monitor disjoint sets of packets at different hosts [47]. We divide the measurement module into two components, namely a *normal path* and a *fast path*. The normal path deploys one or multiple sketch-based solutions as chosen by network operators, while the fast path complements the normal path by deploying a fast but slightly less accurate measurement algorithm to process packets under high traffic load. Normally, the software switch forwards all packets to the normal path through a bounded *FIFO buffer*, which can hold all packets to be processed and absorb any transient spike. However, when the traffic load exceeds the processing capacity of the normal path, the buffer becomes full. In this case, SketchVisor instructs the software switch to redirect *overflowed* packets to the fast path, which then collects traffic statistics from the overflowed packets. We do not consider any proactive approach that examines packets and decides which packets should be dispatched into either the normal path or the fast path, as it will incur non-trivial overhead.

We emphasize that the fast path cannot substitute the sketch-based solutions in the normal path. The main reason is that the fast path is less accurate than the normal path by design. To achieve highly accurate measurement, the normal path has to process as many packets as possible, while the less accurate fast path is activated only when necessary.

SketchVisor's design leaves the deployment decision of what sketch-based solutions should be deployed to network operators, since no sketch-based solution can absolutely outperform others in all aspects. Based on deployment requirements, network operators can choose either a general sketch-based solution (e.g., UnivMon) that supports multiple measurement tasks, or a customized one with better performance for a specific measurement task.

**Challenges:** The FIFO buffer provides a lightweight means to determine when to redirect traffic to the fast path (i.e., by checking if the buffer is full), without compromising the overall measurement performance. The trade-off is that we cannot control which specific flows should have packets sent to the fast path, since tracking specific flows would add processing overhead. This uncertainty complicates the design of the fast path. Also, instead of assigning a fast path per measurement task, we associate a single fast path with all measurement tasks, so that the fast path remains lightweight regardless of how sketches in the normal path are designed. To summarize, the fast path should satisfy the following properties: (*i*) fast enough to absorb all redirected traffic, (*ii*) highly accurate, although the accuracy may slightly degrade from original sketch-based measurement, and (*iii*) general for various traffic statistics.

## 3.2 Control Plane

The control plane (Figure 3(b)) provides a ''one-big-switch" abstraction for network operators to specify and configure measurement tasks at network-wide scale. It collects local measurement results from multiple hosts and merges them to provide network-wide measurement results. Its goal is to achieve accurate network-wide measurement as if *all traffic were only processed by the normal path of each host.*

**Challenges:** It is critical to eliminate the extra errors due to fast path measurement; in other words, all measurement errors should only come from sketches themselves. However, such error elimination heavily hinges on the fast path design, which must be general to accommodate various measurement tasks (§3.1). Similarly, the error elimination in the control plane must be applicable for any measurement task.

## 3.3 Our Solutions

To address the aforementioned challenges, we propose two algorithmic solutions that build on well-studied techniques: the first one builds on counter-based algorithms [15, 33] to design a lightweight, accurate, and general fast path in the data plane (see §4 for details), while the second one builds on compressive sensing [6, 7, 9, 61] to design an accurate network-wide recovery algorithm in the control plane (see §5 for details). We point out that bundling existing techniques directly into SketchVisor does not work as expected. Instead, we carefully analyze the overhead of the existing techniques, and then motivate and design our customized solutions in the context of sketch-based network measurement.

# 4 FAST PATH

## 4.1 Key Idea

The fast path is critical for the robustness of sketch-based measurement. Without the fast path, the normal path unavoidably discards

traffic to keep pace with high traffic load, which compromises measurement accuracy and even makes some measurement tasks fail to work (§7.3).

We design the fast path to track as much information as possible in network traffic with low computational overhead. It is well-known that network traffic in practice exhibits heavy-tailed patterns and is dominated by a few large flows [54, 59], so we expect that the traffic redirected to the fast path is also dominated by large flows (§7.5). Note that this heavy-tailed assumption induces many new sketch designs (e.g., identifying large flows in skewed network traffic). While the inherent sketch designs do not depend on any input distribution, they often achieve better performance under skewed distributions as shown by theoretical analysis [19] and empirical studies [12]. This motivates us to specifically track the largest flows, or top-$k$ flows, in the fast path, where $k$ is configurable depending on the available memory space.

However, tracking only top-$k$ flows is insufficient, since it will inevitably miss information of small flows, which are also critical for connectivity-based statistics (e.g., DDoS, superspreader, and cardinality). Clearly, tracking all small flows in the fast path is infeasible, as the CPU and memory overheads become expensive. Fortunately, sketch-based solutions map flows to counters and leverage the counters to estimate various flow statistics. Our observation is that the values of sketch counters contributed by small flows are generally small and also have low variance when compared to large flows. Thus, we only need to track the overall characteristics of small flows instead of their individual flow headers and sizes. Specifically, we employ a global variable to track the total byte count of these flows, and use it to infer the specific sketch counter values later in the control plane.

**Solution overview:** To this end, we design a fast and accurate top-$k$ algorithm for our fast path. Our algorithm builds on Misra-Gries's top-$k$ algorithm [33]. However, Misra-Gries's algorithm has two limitations that prohibit high performance and accuracy. First, in order to kick out a small flow and add a (potentially) large flow, it performs $O(k)$ operations to update $k$ counters in a hash table; the overhead becomes significant when there are many small flows to kick out. Second, it has loose bounds on the estimated values of the top-$k$ flows. To overcome both limitations, we combine the idea of probabilistic lossy counting (PLC) [15], a probabilistic algorithm that improves accuracy for tracking skewed data, with Misra-Gries's algorithm. Specifically, we kick out multiple small flows each time, obviating the need of performing $O(k)$ counter update operations for kicking out each flow (i.e., we amortize the operations over multiple kick-outs). Also, instead of using one counter per flow, we carefully associate *three* counters with each flow to provide tight per-flow lower and upper bounds.

## 4.2 Algorithm

**Data structure:** We maintain a hash table $H$ that maps flow headers (hash keys) to counters (hash values). We configure $H$ to hold at most $k$ flows. Each flow $f$ is associated with three counters.

- $e_f$ : the maximum possible byte count that can be missed before $f$ is inserted.
- $r_f$ : the residual byte count of $f$.
- $d_f$ : the decremented byte count after $f$ is inserted.

**Algorithm 1** Fast Path Algorithm

**Input**: packet $(f, v)$

1: **function** ComputeThresh($a_1, a_2, \cdots, a_{k+1}$)
2: Find the largest two values $a_1$ and $a_2$ and the smallest value $a_{k+1}$
3: Compute $\theta = \log_b(\frac{1}{2})$, where $b = \frac{a_1-1}{a_2-1}$
4: Return $\hat{e} = \sqrt[\theta]{1-\delta}\, a_{k+1}$ for some small $\delta$
5: **procedure** UpdateBucket($f, v$)
6: $V = V + v$
7: **if** $f$ has an entry $(e_f, r_f, d_f)$ in $H$ **then**
8: Update the entry with $(e_f, r_f + v, d_f)$
9: **else if** $H$ is not full **then**
10: Insert $f$ to $H$ and set $H[f] = (E, v, 0)$
11: **else**
12: $\hat{e}$ = ComputeThresh($\{r_g | g \in H\} \cup \{v\}$)
13: **for all** key $g \in H$ with $H[g] = (e_g, r_g, d_g)$ **do**
14: Update $H[g]$ with $(e_g, r_g - \hat{e}, d_g + \hat{e})$
15: **if** $r_g \leq 0$ **then**
16: Remove $g$ from $H$
17: **if** $v > \hat{e}$ and $H$ is not full **then**
18: Insert $f$ to $H$ and set $H[f] = (E, v - \hat{e}, \hat{e})$
19: $E = E + \hat{e}$

We also keep two global counters for the hash table, which we later use to recover the aggregate statistics of small flows in the control plane (§5).

- $E$ : the sum of all decremented byte counts.
- $V$ : the total byte count of packets in the fast path.

**Algorithm:** Algorithm 1 shows our fast path algorithm. Its idea is to keep the top-$k$ flows in $H$, and remove from $H$ any flow that is below some threshold if $H$ is full. Specifically, upon receiving a packet of size $v$ for flow $f$, we first update the total byte count $V$ (line 6). If $f$ is already in the hash table $H$, we increase the residual byte $r_f$ (lines 7-8); if $H$ is not full (i.e., it has fewer than $k$ flows), we insert $f$ with $(E, v, 0)$ to $H$ (lines 9-10); otherwise, we use ComputeThresh to compute a decremented value $\hat{e}$ (line 12). For each flow $g$ in $H$, we decrease $r_g$ by $\hat{e}$ and increase $d_g$ by $\hat{e}$ (line 14). We kick out flows with residual byte counts no larger than 0 (lines 15-16). We add $f$ to $H$ if its remaining byte count is larger than $\hat{e}$ (lines 17-18). Finally, we update the total decremented byte count $E$ (line 19).

The function ComputeThresh selects a threshold $\hat{e}$ with respect to $k+1$ values, i.e., the values of the top-$k$ flows in $H$ and the value of the new flow $f$. It fits the input values to a power-law distribution and estimates the power-law exponent $\theta$ (line 3) and threshold $\hat{e}$ (line 4), as in PLC [15]. Lemma 4.1 states that Algorithm 1 provides tight lower and upper bounds of each top-$k$ flow tracked by $H$. In Appendix, we explain how $\theta$ and $\hat{e}$ are derived, and present the proof of Lemma 4.1.

Lemma 4.1. *Algorithm 1 has the following properties:*

- *If flow $f$ has size $v_f > E$, it must be tracked in $H$.*
- *If $f \in H$, $r_f + d_f \leq v_f \leq r_f + d_f + e_f$.*
- *For any flow, its maximum possible error is bounded by $O(\frac{V}{k})$.*

**Discussion:** Lemma 4.1 does not assume any statistical distribution. It implies that the per-flow error, which is bounded by $O(\frac{V}{k})$, decreases with $k$. On the other hand, tracking more large flows

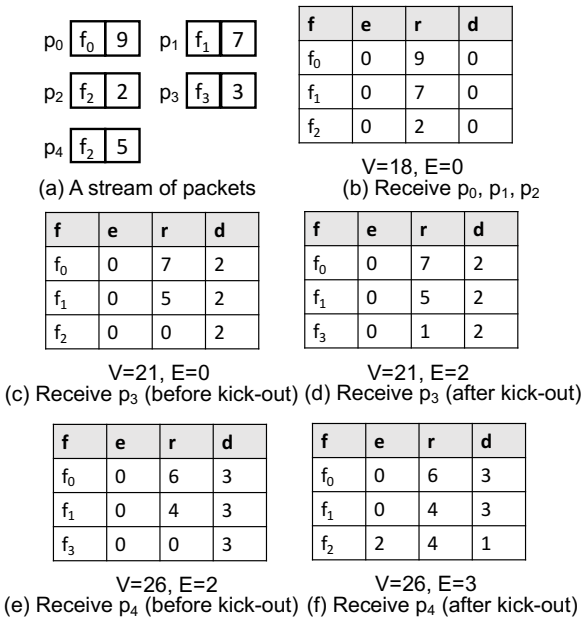



**Figure 4: Example of fast path.**

needs more memory and time to traverse the hash table for kick-out operations. Thus, the value of $k$ trades between performance and accuracy. Nevertheless, a small hash table suffices in practice since the fast path is activated only when necessary (§7.5).

**Example:** We use an example (Figure 4) to illustrate how Algorithm 1 works and how the counters bound the flow sizes. Suppose that we have a stream of five packets and the hash table $H$ has three buckets. For the first three packets (i.e., $p_0, p_1, p_2$), we insert three flows (i.e., $f_0, f_1, f_2$) into $H$, and $V$ represents their total byte count so far (Figure 4(b)). For the fourth packet $p_3$, since $H$ is full, we want to kick out small flows and check if we can insert the new flow $f_3$. To do this, we invoke ComputeThresh to compute a decrement for each flow (which is 2 in this case), and update their $r$ and $d$ (Figure 4(c)). We kick out $f_2$ because its $r$ becomes 0, and insert the new flow $f_3$ (Figure 4(d)). Finally, we see a packet $p_4$ from $f_2$ again. We still use ComputeThresh to kick out a small flow, which is $f_3$ in this case (Figure 4(e)), and insert $f_2$ (Figure 4(f)). Note that we set the $e$ of $f_2$ to be 2 because we have decremented 2 bytes in total before $f_2$ is inserted. This $e$ represents the maximum possible byte count that is not included for $f_2$ when $f_2$ is not in the table yet. Thus, the upper bound of $f_2$ is $e + r + d = 7$. The lower bound is $r + d = 5$ because we count every byte after the flow is inserted. We emphasize that in this example, we kick out one flow each time for brevity, but in general, ComputeThresh computes a threshold that can kick out multiple flows at a time, which is our main improvement over Misra-Gries's algorithm [33].

**Generality:** Our fast path design is applicable for general traffic statistics listed in §2.1. The fast path monitors 5-tuple flows and clearly supports flow-based statistics. It can also extract IP addresses from 5-tuples for host-based statistics. To track more fine-grained flows, we only need to extend the flow definition with more fields (e.g., MAC addresses).

The fast path is volume-based and tracks byte counts, yet we can also use it to track connectivity-based statistics (e.g., DDoS, superspreader, and cardinality) by converting connectivity-based sketches in the normal path into volume-based sketches, similar to the approach in Counting Bloom Filter [4, 34]. Specifically, connectivity-based sketches typically maintain bit arrays and set a bit to one if any observed flow/host is hashed to the bit. We now replace bits by counters and update the counters by byte counts.

# 5 NETWORK-WIDE RECOVERY

## 5.1 Key Idea

The control plane provides network-wide measurement by periodically collecting local measurement results from all hosts and operating on the global views of the normal path and the fast path. Specifically, it aggregates all sketches via matrix additions into a single sketch $N$ (i.e., the sketch counters at the same position are added together), merges all top-$k$ flows and their respective estimated byte counts into a single hash table $H$[1], and adds all recorded total byte counts into $V$. Note that the fast path loses information, as it only holds approximate counters for top-$k$ flows and does not keep track of specific small flows. Thus, given $N$, $H$, and $V$, the goal of the control plane is to accurately recover the missing information and hence the true sketch $T$, as if all traffic were only recorded in $T$.

**Solution overview:** We first formulate the recovery of $T$ as a *matrix interpolation* problem (§5.2). Our formulation also demonstrates the hardness of the recovery problem. To this end, we leverage *compressive sensing* [6, 7, 9, 61] to solve the recovery problem by incorporating domain knowledge into optimization.

## 5.2 Problem Formulation

Interpolation refers to reconstructing missing values based on incomplete and/or indirect observations. In our case, we formulate a matrix interpolation problem that recovers the true sketch $T$ by filling the missing values in $N$ based on $H$ and $V$. We first derive problem constraints that need to be satisfied by $T$.

**Constraints:** We decompose the traffic (in bytes) in the fast path into two $2^{104} \times 1$ vectors indexed by 5-tuple flow header space, namely $\mathbf{x}$ and $\mathbf{y}$, where $\mathbf{x}$ denotes the vector of the actual byte counts of the tracked flows (i.e., flows in $H$) and $\mathbf{y}$ denotes the vector of the actual byte counts of other flows. If a flow does not exist, its vector element has value zero. Thus, the vector $\mathbf{x} + \mathbf{y}$ describes the per-flow traffic counts in the fast path.

To recover $T$, conceivably, we could inject all traffic of the fast path back to the normal path. This in essence applies the sketch function to $\mathbf{x} + \mathbf{y}$, denoted by $\text{sk}(\mathbf{x} + \mathbf{y})$, and adds $\text{sk}(\mathbf{x} + \mathbf{y})$ to $N$ to obtain $T$:

$$T = N + \text{sk}(\mathbf{x} + \mathbf{y}). \quad (1)$$

However, both $\mathbf{x}$ and $\mathbf{y}$ are unknown in practice, as the fast path does not track the actual byte counts of individual flows. Nevertheless, we can specify their constraints. First, the fast path tracks the total byte count $V$. We can relate $\mathbf{x}$ and $\mathbf{y}$ to $V$ via their $l_1$-norms (resp. $\|\mathbf{x}\|_1$ and $\|\mathbf{y}\|_1$) as:

$$\|\mathbf{x}\|_1 + \|\mathbf{y}\|_1 = V. \quad (2)$$

[1]To simplify notation, we overload $H$ to denote the global hash table in this section.

Also, while the merged hash table $H$ does not track $\mathbf{x}$, it gives the lower and upper bounds for each flow due to Lemma 4.1:

$$r_f + d_f \le x_f \le r_f + d_f + e_f. \quad (3)$$

**Hardness:** Our problem is to find $T$ that satisfies the constraints Equations (1)-(3), in which Equations (1) and (2) characterize the aggregate properties of the traffic in the fast path, while Equation (3) quantifies the errors of individual flows. Unfortunately, the fast path only provides incomplete information, and the above constraints are insufficient to unambiguously determine $T$; instead, there exist multiple feasible solutions. This so-called *underconstrained* problem is commonly found in many matrix interpolation problems [24, 60]. Thus, instead of finding a closed form of $T$, we find the ''best'' estimate of $T$.

## 5.3 Compressive Sensing

We leverage compressive sensing [6, 7, 9, 61] to solve our underconstrained matrix interpolation problem. Compressive sensing provides a framework to integrate domain knowledge about matrix structures, so as to eliminate feasible but irrelevant solutions and form a solvable optimization problem [9, 61]. In our case, we incorporate our knowledge about the properties of network traffic and sketches to form an appropriate optimization objective function.

**Properties:** We first identify the properties for $T$, $\mathbf{x}$, and $\mathbf{y}$.

- $T$ **is approximated as a low-rank matrix:** As network traffic is dominated by large flows [54, 59], few counters in $T$ have much different values from other counters that are only accessed by small flows. Thus, we can approximate $T$ as a low-rank matrix (see justifications later).
- **Both** $\mathbf{x}$ **and** sk($\mathbf{x}$) **are sparse:** Since $\mathbf{x}$ only includes the top flows in $H$ and the entire flow space has a very large size (e.g., $2^{104}$ for 5-tuple flows), we can treat $\mathbf{x}$ as a sparse vector. Also, each flow in $\mathbf{x}$ touches a limited number of counters in a sketch, so sk($\mathbf{x}$) is also sparse.
- **Both** $\mathbf{y}$ **and** sk($\mathbf{y}$) **are of small noise:** Network traffic is often dominated by few large flows that are recorded in $\mathbf{x}$. The remaining flows in $\mathbf{y}$ are all very small and their sizes have low variance. Thus, we can treat $\mathbf{y}$ as a small-noise vector. In addition, a sketch maps such small-noise flows uniformly to its counters, so $sk(\mathbf{y})$ is also of small noise.

Before describing how we incorporate the above properties into an optimization objective, we conduct rank analysis to validate the low-rank approximation of $T$. We apply singular value decomposition to generate low rank approximations [18] for several sketch matrices, using the same configurations in §7. Figure 5 shows the relative errors (measured by Frobenius norm) of the low rank approximations. Reversible Sketch [46], Deltoid [13], and TwoLevel [56] take only around 50%, 32%, and 15% of singular values to achieve low rank approximations with less than 10% of errors, respectively (i.e., they can capture more than 90% of information). On the other hand, the relative error of Count-Min Sketch [14] drops linearly with the ratio of top singular values. The reason is that it typically has few rows (less than 10) with thousands of counters each. Such a simple matrix has a rank equal to its number of rows and shows no low rank approximation. Nevertheless, we can still leverage the optimizations of $\mathbf{x}$ and $\mathbf{y}$ to accurately recover $T$.

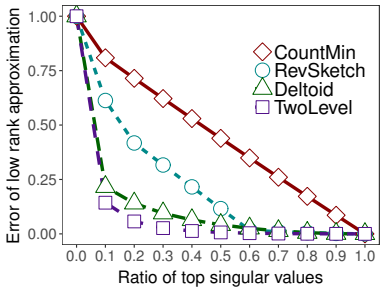



**Figure 5: Error of low rank approximation for sketch-based solutions.**

**Objective function:** We now encode the above properties into the objective function, and leverage the compressive sensing framework LENS [9] to recover $T$. LENS works by decomposing a traffic matrix into low-rank, sparse, and small-noise components and forming an objective function that characterizes the components. Note that LENS mainly addresses traffic matrices that specify traffic volume between all source and destination pairs, while we focus on sketches that map flows to counters and have completely different structures from traffic matrices. Nevertheless, our components $T$, $\mathbf{x}$, and $\mathbf{y}$ actually share similar properties to LENS as argued above. Thus, we follow LENS and derive the following objective function:

$$\text{minimize:} \quad \alpha\|T\|_* + \beta\|\mathbf{x}\|_1 + \frac{1}{2\gamma}\|\mathbf{y}\|_F^2, \quad (4)$$

where $\alpha$, $\beta$, and $\gamma$ are weighting parameters that are configurable (see details below). The three terms in the objective function have the following meanings:

- $\|T\|_* = \sum_i \sigma_i$, where $\sigma_i$'s are singular values of $T$. It is the nuclear norm [44] of $T$ and penalizes against the high rank of $T$.
- $\|\mathbf{x}\|_1 = \sum_i |x_i|$. It is the $l_1$-norm of $\mathbf{x}$ and penalizes against the lack of sparsity in $\mathbf{x}$.
- $\|\mathbf{y}\|_F^2 = \sum_i y_i^2$. It is the squared Frobenius norm of $\mathbf{y}$ and penalizes against large elements in $\mathbf{y}$.

Our objective function provides a general framework for the recovery of all sketches, even though some terms may not be necessary. For example, the term $\mathbf{y}$ has limited impact on sketches for heavy hitter detection, since heavy hitter detection mainly focuses on large flows in $\mathbf{x}$ and a sub-optimal $\mathbf{y}$ is also acceptable. Also, for sketches that do not have low-rank approximations (e.g., Count-Min Sketch in Figure 5), the nuclear norm of $T$ is nearly a constant, so we can discard the term $\|T\|_*$ in the optimization objective.

**Problem solving and parameter settings:** The optimization problem minimizes the objective function Equation (4) subject to the constraints Equations (1)-(3). This is a convex optimization problem, which is computationally tractable. We use the Alternative Direction Method to efficiently solve this problem [9]. Our optimization formulation has three parameters, i.e., $\alpha$, $\beta$, and $\gamma$. Following the guidelines of LENS [9], we set parameters as follows.

$$\alpha = (\sqrt{m_T} + \sqrt{n_T})\sqrt{\eta(N)}.$$
$$\beta = \sqrt{2\log(m_x \cdot n_x)} = \sqrt{2 \times 104}.$$
$$\gamma = 10 \cdot \gamma_y.$$

First, we consider $\alpha$. The constants $m_T$ and $n_T$ are the numbers of rows and columns of matrix $T$, while $\eta(N)$ is the probability density of matrix $N$ and is set as $\frac{\sum_{i,j} N[i][j]}{m_T \cdot n_T}$. Next, we consider $\beta$. The

constants $m_x$ and $n_x$ are numbers of rows and columns of vector $\mathbf{x}$, respectively. Since $\mathbf{x}$ is a $2^{104} \times 1$ vector, we have $\beta = \sqrt{2 \times 104}$. Finally, we consider $\gamma_y$. It denotes the measurement noise and is estimated as the standard deviation of vector $\mathbf{y}$.

## 6 IMPLEMENTATION

We have built a prototype of SketchVisor in C that supports various measurement tasks and sketch-based solutions, as summarized in Table 1. All sketches build on the hash function as in `Snort` [50].

**Data plane:** We have implemented SketchVisor's data plane and integrated it with Open vSwitch [41]. It has three components: (*i*) *a kernel module*, which collects and dispatches packets to the normal path and the fast path, (*ii*) *a user-space daemon*, which hosts the normal path, and (*iii*) *a shared memory block*, which hosts both the normal path's FIFO buffer and the fast path, and is accessible by both the kernel module and the user-space daemon.

The kernel module is an extension to the original datapath kernel module of Open vSwitch. When a packet arrives, the kernel module updates the shared memory block, by inserting the packet header to the FIFO buffer or directly updating the fast path if the buffer is full. It also exports a set of interfaces (e.g., `open`, `close`, and `mmap`) to make it accessible by the user-space daemon. Our modification of the datapath module is around 1,400 LOC.

The user-space daemon maintains all required sketches for the normal path, and maps the shared memory block to its own memory space via `mmap`. It continuously reads packet headers from the FIFO buffer and updates sketches. Also, it periodically reports the results of both the normal path and fast path to the control plane every epoch, and resets all counters and variables for the next epoch.

The shared memory block provides a lightweight channel for the kernel module and the user-space daemon to exchange information. It eliminates context switching during measurement, as opposed to the upcall mechanism in Open vSwitch (which is based on Linux NetLink). One challenge is to efficiently synchronize the access to the shared memory block between the kernel module and the user-space daemon. For the FIFO buffer, since it has only a single producer (i.e., the kernel module) and a single consumer (i.e., the user-space daemon), we implement it as *a lock-free circular buffer* that is optimized for cache-line efficiency [27]. For the fast path, the user-space daemon makes a snapshot of the fast path and resets the fast path immediately when reporting results. When it reports the snapshot, the kernel module continues to update the fast path without being blocked.

Note that Open vSwitch's kernel-based packet forwarding module works independently with SketchVisor's measurement components. Thus, SketchVisor can be deployed atop other software packet processing frameworks with high packet forwarding performance (e.g., Open vSwitch integrated with the Data Plane Development Kit (DPDK) [17]). In such environments, we expect that SketchVisor provides even more performance and accuracy benefits, as the sketch-based measurement overhead now becomes more significant; we plan to study this issue in future work.

**Control plane:** The control plane implements network-wide recovery. It receives results from the data plane in each host through `ZeroMQ` [58]. The compressive sensing solver is based on [9, 61], and uses the `svdcomp` [53] library for singular value decomposition.

| Measurement task | Sketch-based solutions |
|---|---|
| Heavy hitter (HH) detection | FlowRadar [28] |
| | RevSketch [46] |
| | UnivMon [30] |
| | Deltoid [13] |
| Heavy changer (HC) detection | FlowRadar [28] |
| | RevSketch [46] |
| | UnivMon [30] |
| | Deltoid [13] |
| DDoS detection | TwoLevel [56] |
| Superspreader (SS) detection | TwoLevel [56] |
| Cardinality estimation | FM [20] |
| | kMin [2] |
| | Linear Counting (LC) [55] |
| Flow size distribution | FlowRadar [28] |
| | MRAC [26] |
| Entropy estimation | FlowRadar [28] |
| | UnivMon [30] |

**Table 1: Measurement tasks and sketch-based solutions.**

## 7 EVALUATION

We conduct experiments to demonstrate that SketchVisor can: (*i*) achieve both high performance and high accuracy for various measurement tasks, (*ii*) work seamlessly with various sketch-based solutions, (*iii*) scale to a large number of hosts in stress tests, and (*iv*) achieve comparable performance with much less memory to [38], a recently proposed measurement framework based on simple hash tables.

### 7.1 Methodology

**Testbed:** We deploy SketchVisor on a testbed composed of nine hosts, each of which is equipped with Intel Xeon X5670 2.93GHz CPU, 300GB memory, a Broadcom BCM5709 NetXtreme Gigabit Ethernet NIC, and a Mellanox MT27710 10-Gigabit Ethernet NIC. We run the data plane in eight hosts, which send traffic through the 10Gb NICs, and the control plane in the remaining host, which communicates with the data plane through the 1Gb NICs. In each host, we run SketchVisor (either data plane or control plane) as a single-threaded process on a dedicated CPU core.

**In-memory tester:** Our testbed is inadequate for scalability evaluation, as its scale is limited by the per-host NIC speed (10Gbps) and the number of physical hosts in the data plane (eight hosts). Thus, we also evaluate a SketchVisor variant called the *in-memory tester*, which executes the core data plane and control plane logics entirely in memory. For the data plane, the in-memory tester processes traffic that is loaded into memory in advance, without forwarding traffic to Open vSwitch and NIC; for the control plane, the in-memory tester performs network-wide recovery from the local measurement results that are again loaded into memory in advance. We run the in-memory tester as a single-threaded process on a dedicated CPU core. Our in-memory tester eliminates network transfer overhead, so as to stress-test the computational performance of SketchVisor.

**Parameter settings:** By default, we allocate 8KB memory for the fast path (we study different fast path sizes in §7.5), and set the parameters of our network-wide recovery algorithm as described in §5. For the measurement tasks and sketch-based solutions in

Table 1, we set their parameters such that the sketch-based solutions for each measurement task have the same error bound based on their theoretical analysis. For some sketch-based solutions (e.g., FlowRadar [30]) that address the worse-case scenario and require excessive resources, we manually reduce their memory usage without increasing their errors based on our experiments.

- *Heavy hitter (HH) detection:* We set the HH threshold as 0.05% of the NIC capacity multiplied by the epoch length. We evaluate four sketches. (*i*) Deltoid: we use four rows with 2/0.05% = 4,000 counters each, and the error probability is $1/2^4 = 1/16$. (*ii*) Reversible Sketch (RevSketch): similar to Deltoid, we use four rows with 4,000 counters each, and partition a 104-bit five tuple into 16-bit words. (*iii*) UnivMon: we allocate 4,000, 2,000, 1,000, 500 counters in the first, second, third sketches, and others, respectively, and track top 500 flows in its heap. (*iv*) FlowRadar: we use four hash functions in both the Bloom Filter and counter array, and set the Bloom Filter length as 100,000 and the counter array length as 40,000.
- *Heavy changer (HC) detection:* We set the threshold as 0.05% of total changes over two adjacent epochs, and use the same sketch settings as in HH detection.
- *DDoS detection:* We set the threshold as 0.5% of the total number of IP addresses. We evaluate TwoLevel [56], which consists of a Count-Min sketch and a RevSketch. For the Count-Min sketch, we allocate two rows with 4,000 counters each, and for each bucket in the Count-Min sketch, we allocate two rows with 250 counters each. For the RevSketch, we allocate two rows with 4,096 counters each to track candidate IP addresses, and partition a 32-bit IP address into four 8-bit words.
- *Superspreader (SS) detection:* We use the same setting as DDoS detection.
- *Cardinality estimation:* We evaluate FM, kMin, and Linear Counting (LC). We allocate four rows with 65,536 counters each for FM and kMin, and four rows with 10,000 counters each for LC.
- *Flow size distribution:* We evaluate MRAC and FlowRadar. For MRAC, we allocate a single row with 4,000 counters; for FlowRadar, we use the same setting as in HH detection.
- *Entropy estimation:* We evaluate FlowRadar and UnivMon with the same setting as in HH detection.

**Workloads:** We use five one-hour public traffic traces collected in 2015 from CAIDA [5]. In our testbed experiments, we evenly partition the traces and distribute them across hosts. We modify the MAC addresses of packets, and replay and forward the packets across hosts. Before each experiment, we load the traces into memory to eliminate any disk IO overhead. Each host sends out traffic as fast as possible to test the maximum throughput of SketchVisor. In practice, the network utilization is often lower, so a higher portion of network traffic can be handled by the normal path and we expect to see better performance and accuracy.

We evaluate each sketch-based solution separately to show its performance gain with SketchVisor; we do not explicitly evaluate the combination of multiple sketch-based solutions, yet some of them (e.g., TwoLevel) comprise multiple sketches by design. The data plane reports measurement results to the controller in one-second epochs. In each epoch, we find that each host generates around 30K-70K flows, 370K-480K packets, and 260MB-330MB traffic. We repeat each experiment 10 times and report the average of all trials across all epochs. We find that the standard error of each trial is insignificant and only deviates from the average by at most 5%, so we omit error bars in our plots.

**Metrics:** We consider the following metrics:

- Throughput: the total traffic volume processed per second (it can be transformed into the packet rate, as the average packet size in our dataset is 769 bytes).
- Recall: the ratio of true instances reported.
- Precision: the ratio of reported true instances.
- Relative error: $\frac{1}{n}\sum_{i=0}^{n-1}\frac{|v_i-\hat{v}_i|}{v_i}$ where $v_i$ is the true value of $i$ and $\hat{v}_i$ is the estimate of $i$.
- Mean Relative Difference (MRD): $\frac{1}{z}\sum_{i=1}^{z}\frac{|n_i-\hat{n}_i|}{(n_i+\hat{n}_i)/2}$, where $z$ is the maximum flow size, and $n_i$ and $\hat{n}_i$ are the true and estimated numbers of flows with size $i$, respectively.

Throughput is used for all tasks, while the remaining metrics are accuracy-related and are used based on the nature of the traffic statistics:

- HH, HC, DDoS, SS: recall, precision, relative error.
- Cardinality, entropy: relative error.
- Flow size distribution: MRD.

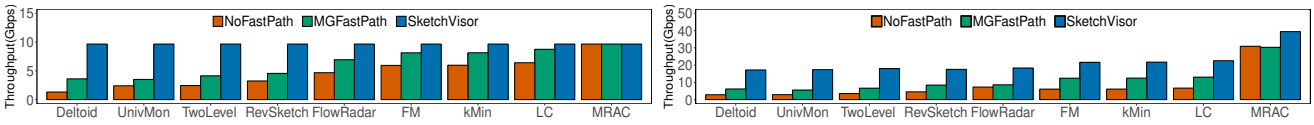
## 7.2 Throughput

We evaluate the throughput of SketchVisor by deploying different sketch-based solutions in the normal path. We compare three alternatives: (*i*) *NoFastPath*, which only executes the normal path without the fast path, (*ii*) *MGFastPath*, which uses the original Misra-Gries's top-$k$ algorithm in the fast path, (*iii*) *SketchVisor*, which uses our proposed top-$k$ algorithm in the fast path.

Figure 6(a) shows the testbed results. SketchVisor achieves almost 10Gbps for all sketches. NoFastPath only achieves almost 10Gbps for MRAC and from 1.32Gbps to 6.41Gbps for others. MGFastPath is faster than NoFastPath, but still achieves no more than 5Gbps for four out of nine sketches. Figure 6(b) shows the in-memory tester results. NoFastPath and MGFastPath still cannot achieve 10Gbps for most sketches. In contrast, SketchVisor achieves over 17Gbps for all sketches, and almost 40Gbps for MRAC in particular. Note that this result is measured in a single CPU core and is much higher than five-core results (without the fast path) in §2. We can further boost the throughput by parallelizing the normal path and fast path with multiple CPU cores and merging their results later in the control plane. Our results show that two CPU cores are sufficient to achieve above 40Gbps for all sketches (not shown in the figure).

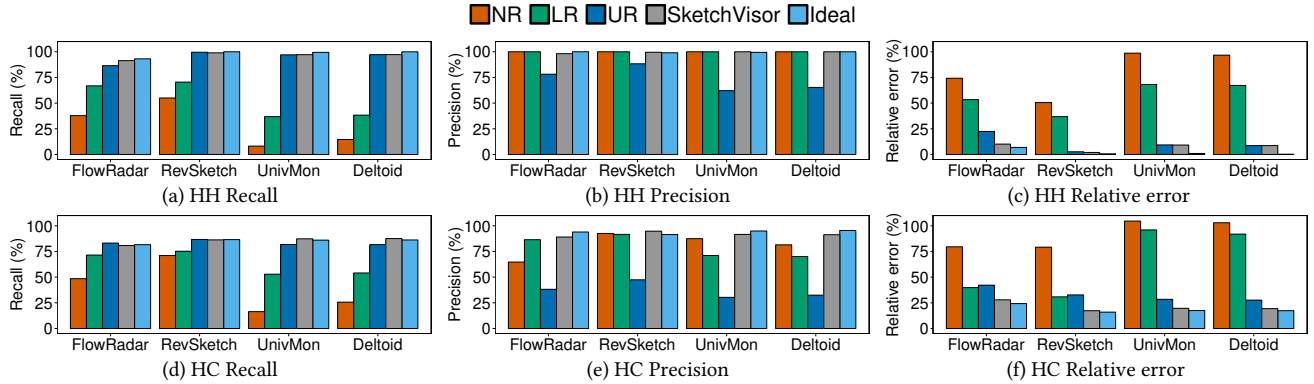
## 7.3 Accuracy

We evaluate the accuracy of SketchVisor. We compare five alternatives: (*i*) *NoRecovery (NR)*, which only uses the normal path results and discards the fast path results, (*ii*) *LowerRecovery (LR)*, which only combines the lower-bound estimates in the fast path with the normal path results, (*iii*) *UpperRecovery (UR)*, which only combines the upper-bound estimates in the fast path with the normal path results, (*iv*) *SketchVisor*, and (*v*) *Ideal*, which uses the normal path to process all traffic, without adding extra errors due to the fast path (i.e., all errors come from sketches themselves). To compute the




(a) Testbed result (b) In-memory tester result

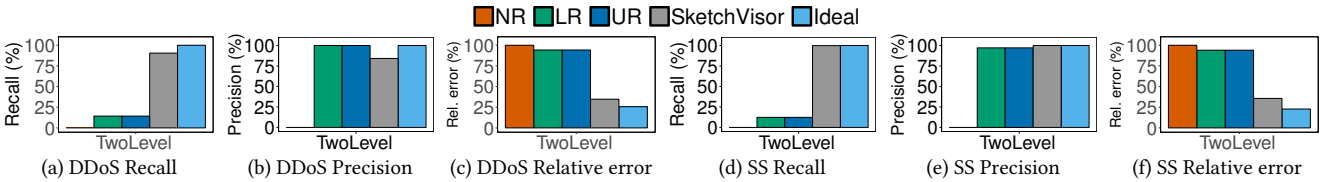
**Figure 6: Throughput of different sketch-based solutions.**




(a) HH Recall (b) HH Precision (c) HH Relative error

(d) HC Recall (e) HC Precision (f) HC Relative error

**Figure 7: Accuracy of HH/HC detection.**




(a) DDoS Recall (b) DDoS Precision (c) DDoS Relative error (d) SS Recall (e) SS Precision (f) SS Relative error

**Figure 8: Accuracy of DDoS and SS detection.**

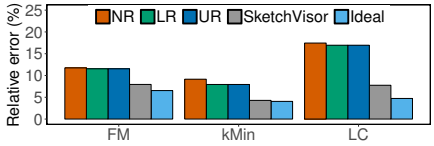



**Figure 9: Cardinality estimation.**

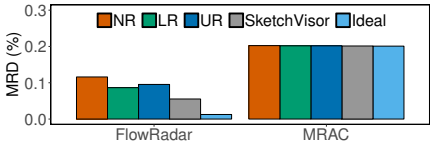



**Figure 10: Flow size distribution.**

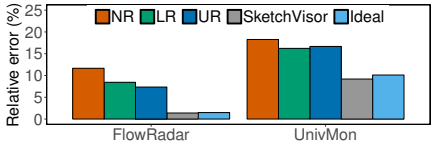



**Figure 11: Entropy estimation.**

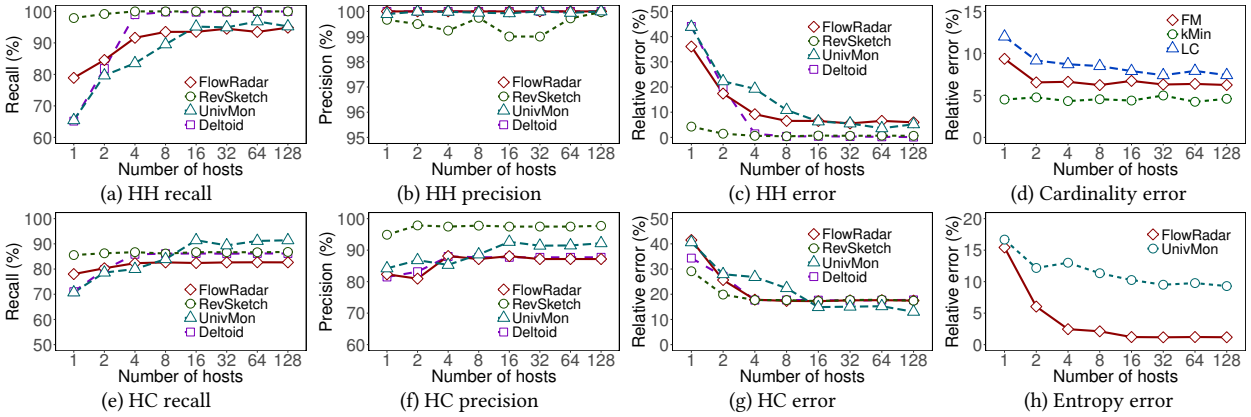
accuracy metrics, we generate the ground truth (with zero error) by tracking the whole trace with a very large hash table, and compare the results of each alternative with the ground truth.

**HH and HC detection (Figure 7):** NR has much lower recall and a higher relative error than Ideal. For example, in UnivMon, the recall of NR is only 8.15% in HH detection and 16.43% in HC detection, while the corresponding relative errors are 98.63% and 102.58%, respectively. The reason is that NR discards all information in the fast path. LR improves the overall recall, but is still below 80% as it underestimates the sizes and changes for many true HHs and HCs. UR achieves high recall, but at a cost of low precision. In contrast, SketchVisor achieves close accuracy to Ideal for all three metrics.

**DDoS and SS detection (Figure 8):** NR, LR, and UR all have low recall and high relative errors. In particular, NR even cannot detect any DDoS or superspreader. LR and UR have the same detection results since DDoS and SS detection concerns the number of hosts instead of flow size. In contrast, SketchVisor achieves nearly perfect results in SS detection. For DDoS detection, the accuracy drops slightly compared to Ideal, but the recall is still above 90% and the precision is above 84%.

**Cardinality estimation (Figure 9):** In FM and kMin, the errors of NR, LR and UR are all nearly twice those in Ideal, while their errors are around 17% in LC. SketchVisor significantly reduces the errors and is close to Ideal. The reason is that all the three sketches estimate cardinality based on non-zero counters. Since the small hash table in the fast path discards many flows, NR, LR, and UR end up with many zero counters in the sketch, and thus have poor accuracy. In contrast, SketchVisor restores non-zero counters with compressive sensing.

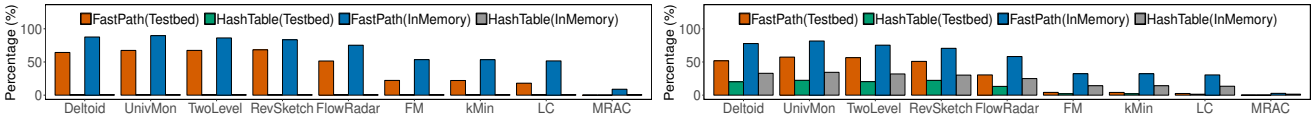
**Flow size distribution (Figure 10):** For MRAC, all approaches achieve near-optimal MRD (around 0.2%), since MRAC is fast enough that only few flows enter the fast path. For FlowRadar, NR, LR, and




(a) HH recall (b) HH precision (c) HH error (d) Cardinality error

(e) HC recall (f) HC precision (g) HC error (h) Entropy error

**Figure 12: Network-wide recovery.**




(a) Percentage of flows (b) Percentage of bytes

**Figure 13: Percentage of traffic in the fast path.**

UR increase the MRD from 0.0126% in Ideal to 0.1166%, 0.0844%, and 0.0954%, respectively, mainly because they do not consider the missing small flows dropped by the fast path. In contrast, SketchVisor reduces the error to 0.0553%.

**Entropy estimation (Figure 11):** Interestingly, SketchVisor has a slightly lower error than Ideal, as it can eliminate a small amount of errors caused by the sketch itself when recovering it using compressive sensing, while Ideal directly returns the sketch that processes all traffic in the normal path.

## 7.4 Network-Wide Recovery

We evaluate the network-wide recovery of SketchVisor. To evaluate a large network size, we use our in-memory tester and configure the control plane to aggregate results from 1 to 128 hosts. Here, we show the results of HH detection, HC detection, cardinality estimation, and entropy estimation in Figure 12. The accuracy results vary across measurement tasks and sketch-based solutions. Overall, SketchVisor improves accuracy as the number of hosts increases. For example, the recall of UnivMon increases from 65% to 81% when the number of hosts increases from one to two. The recall is even above 99% when the number of hosts exceeds four. The reason for accuracy improvement is that integrating results from multiple hosts (*i*) reduces the number of missing values in sketch matrices and (*ii*) increases the number of constraints in our recovery optimization. Also, some sketches (e.g., kMin in cardinality estimation) already achieve high accuracy in a single host, and maintain high accuracy as the number of hosts increases.

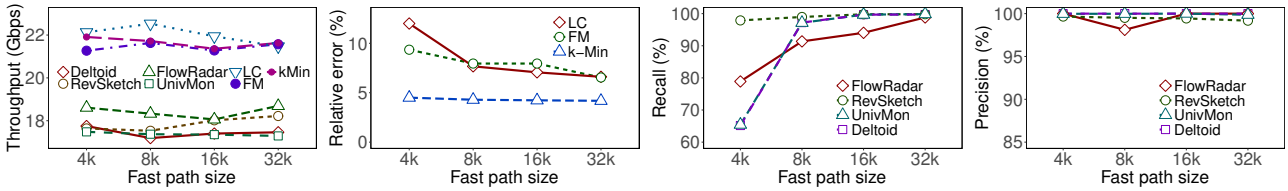
## 7.5 Microbenchmarks

**Percentage of traffic in the fast path:** Figure 13(a) shows that SketchVisor redirects more than 20% (resp. 50%) of flows to the fast path in the testbed (resp. in-memory tester), except for MRAC. Figure 13(b) shows that the fast path processes more than 50% of byte counts for most tasks in both testbed and in-memory experiments. The percentage for MRAC is negligible since MRAC is a simple sketch. Note that our default 8KB fast path only records around 0.7% of total flows (Figure 13(a)), while contributing to over 20% of byte counts (Figure 13(b)) due to traffic skewness.

We further examine the traffic redirected to the fast path specifically, and find that the top 10% of flows tracked by the fast path account for over 90% of byte counts for all solutions except MRAC, and over 80% of byte counts for MRAC; we do not plot the results in the interest of space.

**Impact of fast path size:** We configure various sizes for the fast path: 4KB, 8KB, 16KB, and 32KB. We measure the throughput and accuracy: HH and cardinality, using the same accuracy metrics in §7.3. Figure 14(a) shows that the throughput varies by less than 5% across fast path sizes. The reason is that while a larger hash table in the fast path implies a longer time to search for small flows to be kicked out, it also sustains more hash table insertions/updates before triggering a new kick-out operation. Figures 14(b)-(d) show the accuracy versus the fast path size. The accuracy improves remarkably when the fast path size increases from 4KB to 8KB (e.g., the HH recall of Deltoid increases from 65.17% to 97.21%), and stabilizes when the fast path size exceeds 8KB.

**Computation time of network-wide recovery:** The computation time to solve compressive sensing varies from 0.15 seconds (for MRAC) to 64 seconds (for Deltoid) in a single CPU core, depending on the number of sketch counters (we omit the figures in the interest of space). We can reduce the computation time in two ways. First, some terms in the objective function do not need to be optimal for some sketches (see discussion in §5.3), so it is possible to terminate the computation early even though these unnecessary terms do not converge. We have evaluated this optimization and




(a) Throughput (b) Cardinality error (c) HH Recall (d) HH Precision

**Figure 14: Impact of the fast path size.**

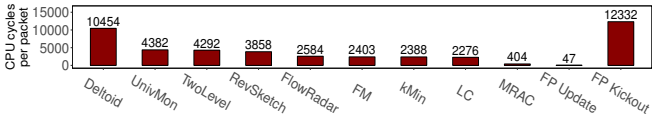
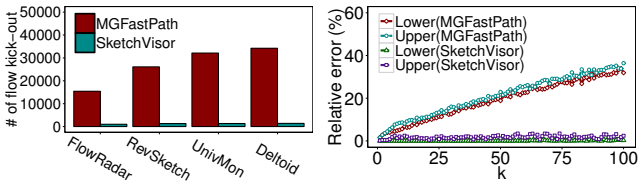
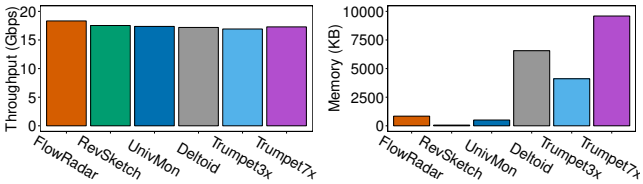



**Figure 15: CPU overhead of sketch-based solutions.**




(a) Number of flow kick-outs (b) Errors of top-k flows

**Figure 16: Comparison with MGFastPath.**




(a) Throughput (b) Memory consumption

**Figure 17: Comparison with Trumpet [38].**

find that the computation time for Deltoid can decrease from 64 seconds to 11 seconds. Second, our current solver is single-threaded. Since the recovery across epochs is independent, we can parallelize network recovery through multiple CPU cores.

**Comparison with Misra-Gries's algorithm:** We compare the fast path of SketchVisor with the original Misra-Gries's algorithm (MGFastPath). We consider two metrics: (*i*) the number of kick-out operations, which accounts for the major overhead in the fast path, and (*ii*) the relative errors of top flows (including both lower and upper bounds). Figure 16(a) shows that MGFastPath performs an order of magnitude more kick-out operations than SketchVisor. This explains why MGFastPath only slightly improves the throughput compared to NoFastPath (Figure 6). Figure 16(b) shows the relative errors on both lower bounds and upper bounds of top-$k$ flows in the fast path for Deltoid. MGFastPath increases the errors as $k$ grows. For the 100-th flow, the relative error increases to 35%. In contrast, SketchVisor keeps the errors under 2% as we tighten the lower and upper bounds using three counters per flow.

**CPU overhead for normal path and fast path:** We revisit the CPU overhead of different sketch-based solutions when the fast path is used. Figure 15 shows the number of CPU cycles for recording a packet in each sketch-based solution, as well as those for the update and kick-out operations of the fast path. The number of CPU cycles varies across sketch-based solutions, from 404 (for MRAC) to 10,454 (for Deltoid). In contrast, the fast path spends only 47 cycles to record a new flow or update an existing flow in its hash table. While a kick-out incurs excessive CPU overhead, the fast path limits the number of kick-outs (Figure 16(a)).

## 7.6 Comparison with Trumpet

Finally, we show that SketchVisor can approach the performance and accuracy of simple hash tables [1] (§2.2), while using much less memory. We consider the recently proposed Trumpet [38], a software measurement architecture that tracks per-flow information in simple hash tables rather than sketches. Specifically, we implement Trumpet Packet Monitor to monitor traffic in the data plane, and deploy a single trigger to monitor heavy hitters. This trigger requires a single variable for byte counts and does not contain any predicates to filter traffic. Note that Trumpet deals with hash collisions by over-provisioning hash tables, but requires substantial memory to completely eliminate collisions. Therefore, our implementation allocates a hash table with a small over-provisioning factor and deals with hash collisions by linked lists. We present the results with over-provisioning factors 3 and 7, referred to as Trumpet3x and Trumpet7x, respectively.

Figure 17 compares the throughput and memory consumption of SketchVisor with Trumpet. SketchVisor achieves similar throughput as Trumpet (Figure 17(a)). However, the sketches except Deltoid consume much less memory than Trumpet (Figure 17(b)). The reason is that Trumpet tracks per-flow information in a hash table, while sketches store information in a fixed number of counters. Although Trumpet provides perfect monitoring, we have shown that sketches can also achieve near-optimal accuracy for various tasks. Thus, SketchVisor provides an efficient alternative for network measurement, especially when the hash table size increases linearly with the number of flows.

# 8 RELATED WORK

Our work is related to software-defined measurement. We review related work in this area.

**Sampling:** Sampling is widely used in software-defined measurement for low measurement overhead. Sekar et al. [48] combine flow sampling and sample-and-hold [19] as primitives for various measurement applications. OpenSample [52] reconstructs flow statistics based on sampled traffic. Planck [43] mirrors traffic to remote sites in a best-effort manner. However, sampling inherently misses information and supports only coarse-grained measurement.

**Sketches:** Many architectures employ sketches as primitives to achieve fine-grained measurement for various measurement tasks

(see Table 1). In the context of software-defined measurement, OpenSketch [56] defines APIs for general sketch-based measurement tasks running in commodity switches. SCREAM [37] addresses dynamic resource allocation of sketch-based measurement across multiple switches. However, sketch-based measurement incurs high computational overhead as shown in our analysis (§2). Although we can deploy distributed sketch-based measurement [11, 22] to boost performance, it still needs excessive computational resources for parallelization.

**TCAM:** TCAM can be used to achieve high-performance network measurement. Jose et al. [23] propose a TCAM measurement framework based on OpenFlow [31]. DREAM [36] dynamically allocates TCAM for high measurement accuracy. PathQuery [39] monitors path-level traffic with TCAM. In contrast, our work address software packet processing without specific hardware support.

**Rule matching:** Rule matching selectively processes only packets of interest, thereby reducing measurement overhead. ProgME [57] and EverFlow [62] filter flows based on pre-defined rules. NetSight [21] leverages SDN to capture packets for specific forwarding events. MOZART [29] and Trumpet [38] monitor network-wide events with hash tables to achieve high throughput [1], by matching flows to events and storing only matched flows in hash tables. However, hash-table-based measurement incurs much higher memory overhead than sketch-based measurement (§7.6). Note that rule matching requires careful configuration of matching criteria to avoid compromising measurement accuracy.

## 9 CONCLUSION

We design and implement SketchVisor, a robust network-wide measurement architecture for software packet processing, with a primary goal of preserving performance and accuracy guarantees even under high traffic load. SketchVisor employs sketches as basic measurement primitives, and achieves high data plane performance with a fast path to offload sketch-based measurement under high traffic load. It further leverages compressive sensing to achieve accurate network-wide measurement. Experiments demonstrate that SketchVisor achieves high performance and high accuracy for a rich set of sketch-based solutions.

## ACKNOWLEDGMENTS

We thank our shepherd, Dina Papagiannaki, and the anonymous reviewers for their valuable comments. We thank Jennifer Rexford for comments on the earlier version of the paper.

## REFERENCES

[1] O. Alipourfard, M. Moshref, and M. Yu. Re-evaluating Measurement Algorithms in Software. In *Proc. of HotNets*, 2015.
[2] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. In *Proc. of RANDOM*, 2002.
[3] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of IMC*, 2010.
[4] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An Improved Construction for Counting Bloom Filters. In *Proc. of ESA*, 2006.
[5] Caida Anonymized Internet Traces 2015 DCaida. http://www.caida.org/data/passive/passive_2015_dataset.xml.
[6] E. Candes, J. Romberg, and T. Tao. Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on Information Theory*, 52(2):489--509, 2006.
[7] E. Candes and T. Tao. Near-Optimal Signal Recovery From Random Projections: Universal Encoding Strategies? *IEEE Transactions on Information Theory*, 52(12):5406--5425, 2006.
[8] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. *Theoretical Computer Science*, 312(1):3--15, 2004.
[9] Y.-C. Chen, L. Qiu, Y. Zhang, G. Xue, and Z. Hu. Robust Network Compressive Sensing. In *Proc. of MOBICOM*, 2014.
[10] Cisco Nexus 1000V Switch. http://www.cisco.com/c/en/us/products/switches/nexus-1000v-switch-vmware-vsphere/index.html.
[11] G. Cormode and M. Garofalakis. Sketching Streams Through the Net: Distributed Approximate Query Tracking. In *Proc. of VLDB*, 2005.
[12] G. Cormode and M. Hadjieleftheriou. Methods for Finding Frequent Items in Data Streams. *The VLDB Journal*, 19(1):3----20, 2010.
[13] G. Cormode and S. Muthukrishnan. What's New: Finding Significant Differences in Network Data Streams. In *Proc. of IEEE INFOCOM*, 2004.
[14] G. Crmode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58--75, 2005.
[15] X. Dimitropoulos, P. Hurley, and A. Kind. Probabilistic Lossy Counting: An Efficient Algorithm for Finding Heavy Hitters. *ACM SIGCOMM Computer Communication Review*, 38(1):5--5, 2008.
[16] M. Dobrescu, N. Egi, K. Argyraki, B.-g. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks : Exploiting Parallelism to Scale Software Routers. In *Proc. of SOSP*, 2009.
[17] DPDK. http://dpdk.org/.
[18] C. Eckart and G. Young. The Approximation of One Matrix by Another of Lower Rank. *Psychometrika*, 1(3):211--218, 1936.
[19] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting : Focusing on the Elephants , Ignoring the Mice. *ACM Trans. on Computer Systems*, 21(3):270--313, 2003.
[20] P. Flajolet and G. Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182--209, 1985.
[21] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Proc. of NSDI*, 2014.
[22] Q. Huang and P. P. C. Lee. A Hybrid Local and Distributed Sketching Design for Accurate and Scalable Heavy Key Detection in Network Data Streams. *Computer Networks*, 91:298--315, 2015.
[23] L. Jose, M. Yu, and J. Rexford. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *USENIX HotICE*, 2011.
[24] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *Proc. of IMC*, 2009.
[25] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proc. of NSDI*, 2014.
[26] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. In *Proc. of SIGMETRICS*, 2004.
[27] P. P. C. Lee, T. Bu, and G. Chandranmenon. A Lock-Free , Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring. In *Proc. of IPDPS*, 2010.
[28] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *Proc. of NSDI*, 2016.
[29] X. Liu, M. Shirazipour, M. Yu, and Y. Zhang. MOZART: Temporal Coordination of Measurement. In *Proc. of SOSR*, 2016.
[30] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. of SIGCOMM*, 2016.
[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69, 2008.
[32] Microsoft Hyper-V Virtual Switch. https://technet.microsoft.com/en-us/library/hh831823.aspx.
[33] J. Misra and D. Gries. Finding Repeated Elements. *Science of Computer Programming*, 2(2):143--152, 1982.
[34] M. Mitzenmacher. Compressed Bloom Filters. In *Proc. of PODC*, 2001.
[35] M. Mitzenmacher, R. Pagh, and N. Pham. Efficient Estimation for High Similarities Using Odd Sketches. In *Proc. of WWW*, 2014.
[36] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. of SIGCOMM*, 2014.
[37] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *Proc. of CoNEXT*, 2015.
[38] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *Proc. of SIGCOMM*, 2016.
[39] S. Narayana, M. T. Arashloo, J. Rexford, and D. Walker. Compiling Path Queries. In *Proc. of NSDI*, 2016.
[40] NetFlow. https://www.ietf.org/rfc/rfc3954.txt.
[41] OpenvSwitch. http://openvswitch.org.

[42] perf. http://perf.wiki.kernel.org.
[43] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *Proc. of SIGCOMM*, 2014.
[44] B. Recht, M. Fazel, and P. A. Parrilo. Guaranteed Minimum-Rank Solutions of Linear Matrix Equations via Nuclear Norm Minimization. *SIAM Review*, 52(3):471--501, 2010.
[45] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. of ATC*, 2012.
[46] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. Dinda, M. Y. Kao, and G. Memik. Reversible Sketches: Enabling Monitoring and Analysis over High-Speed Data Streams. *IEEE/ACM Trans. on Networking*, 15(5):1059--1072, 2007.
[47] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. cSAMP: A System for Network-Wide Flow Monitoring. In *Proc. of USENIX NSDI*, 2008.
[48] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *Proc. of IMC*, 2010.
[49] sFlow. http://www.sflow.org/.
[50] Snort. https://www.snort.org.
[51] SPAN. http://www.cisco.com/c/en/us/tech/lan-switching/switched-port-analyzer-span/index.html.
[52] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter. OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN. In *Proc. of ICDCS*, 2014.
[53] svdcomp. http://www.public.iastate.edu/~dicook/JSS/paper/code/svd.c.
[54] K. Thompson, G. J. Miller, and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network*, 11:10--23, 1997.
[55] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A Linear-time Probabilistic Counting Algorithm for Database Applications. *ACM Trans. Database Systems*, 15(2):208--229, 1990.
[56] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *Proc. of NSDI*, 2013.
[57] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards Programmable Network Measurement. In *Proc. of SIGCOMM*, 2007.
[58] ZeroMQ. http://zeromq.org.
[59] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the Characteristics and Origins of Internet Flow Rates. In *Proc. of ACM SIGCOMM*, 2002.
[60] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg. Fast Accurate Computation of Large-Scale IP Traffic Matrices from Link Loads. In *Proc. of SIGMETRICS*, 2003.
[61] Y. Zhang, M. Roughan, W. Willinger, and L. Qiu. Spatio-Temporal Compressive Sensing and Internet Traffic Matrices. In *Proc. of SIGCOMM*, 2009.
[62] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *Proc. of SIGCOMM*, 2015.

## Appendix A: Derivation of $\theta$ and $e$

We derive $\theta$ and $e$ in ComputeThresh of Algorithm 1. Without loss of generality, we denote the input $k+1$ values as $a_1, a_2, \cdots, a_{k+1}$, where $a_1 \geq a_2 \geq \cdots a_{k+1}$. We fit the inputs to a power-law distribution; that is, for some random variable $Y$, we have $\Pr\{Y > y\} = \epsilon y^{\theta}$, where $\epsilon \leq 1$ and $\theta < 0$ are some control parameters. We are only concerned about the exponent $\theta$ for computing $e$, as shown below.

To estimate $\theta$, we only consider the two largest values $a_1$ and $a_2$, which are the most likely to be the actual largest flows among all flows being tracked in the hash table $H$. Their probabilities can be described as $\Pr\{Y > (a_1 - 1)\} = \epsilon(a_1 - 1)^{\theta} \approx \frac{1}{n}$ and $\Pr\{Y > (a_2 - 1)\} = \epsilon(a_2 - 1)^{\theta} \approx \frac{2}{n}$, respectively, where $n$ is the number of total flows. By dividing the first equation by the second one, we have $\frac{(a_1-1)^{\theta}}{(a_2-1)^{\theta}} \approx \frac{1}{2}$. Thus, we estimate $\theta = \log_b(\frac{1}{2})$, where $b = \frac{a_1-1}{a_2-1}$.

To compute $e$, we ensure that it is larger than the smallest value $a_{k+1}$, so that we can always kick out the smallest flow among the $k+1$ inputs (i.e., the top-$k$ flows in the hash table $H$ and the new flow); on the other hand, it cannot be too large to avoid kicking out large flows. Thus, we compute $e$ such that the probability of kicking out any flow that is larger than $a_{k+1}$ is no larger than some small parameter $\delta$ (e.g., we can set $\delta = 0.05$). The probability is calculated as follows:

$$\Pr\{Y \leq e \mid Y > a_{k+1}\} = \frac{\Pr\{a_{k+1} < Y \leq e\}}{\Pr\{Y > a_{k+1}\}}$$

$$= \frac{\Pr\{Y > a_{k+1}\} - \Pr\{Y > e\}}{\Pr\{Y > a_{k+1}\}} = \frac{\epsilon a_{k+1}^{\theta} - \epsilon e^{\theta}}{\epsilon a_{k+1}^{\theta}} = \frac{a_{k+1}^{\theta} - e^{\theta}}{a_{k+1}^{\theta}}.$$

Setting the right side to $\delta$ yields $e = \sqrt[\theta]{1-\delta}\, a_{k+1}$.

## Appendix B: Proof of Lemma 4.1

We prove the three properties of Lemma 4.1 (see §4).

(*i*) *If flow $f$ has size $v_f > E$, it must tracked in $H$.*

Our fast path algorithm decrements a flow by at most $E$ after all flow kick-out operations (lines 14-17 in Algorithm 1). For any flow $f$ with size $v_f > E$, its residual counter $r_f$ is at least $v_f - E$. Since we only kick out a flow when its residual byte count reaches zero, the flow must be tracked in $H$. The result follows.

(*ii*) *If $f \in H$, $r_f + d_f \leq v_f \leq r_f + d_f + e_f$.*

The first inequality holds because $r_f + d + f$ is the exact byte count after $f$ is inserted to $H$. On the other hand, the missing byte count before $f$ is inserted is at most $e_f$. Thus, the total byte count $v_f$ (including the byte counts before $f$ is inserted and after $f$ is inserted) is at most $r_f + d_f + v_f$.

(*iii*) *For any flow, its maximum possible error is bounded by $O(\frac{V}{k})$.*

The error of a flow $f$ is less than $e_f \leq E$. Thus, we just need to prove that $E$ is bounded. Our proof assumes that our parameter estimation yields the same parameter $\theta$, although the actual estimated $\theta$ can vary (slightly).

We prove the result by computing the byte count that is decremented in a kick-out operation. Since we select the threshold $e$ larger than the smallest value $a_{k+1}$, the actual decremented byte count must be larger than $a_{k+1}$. If we use $a_{k+1}$ as the threshold, then the total decremented byte count is exactly $(k+1)a_{k+1} = (k+1)\frac{e}{\sqrt[\theta]{1-\delta}}$. This implies that the actual decremented byte count is at least $(k+1)a_{k+1} = (k+1)\frac{e}{\sqrt[\theta]{1-\delta}}$ in a kick-out operation.

On the other hand, the total decremented byte count in all kick-out operations should not exceed the total byte count $V$. Thus, we have $\sum_e (k+1)\frac{e}{\sqrt[\theta]{1-\delta}} \leq V$. Since $E$ is the sum of $e$ is all kick-out operations (i.e., $E = \sum_e e$), we have $E \leq \sqrt[\theta]{1-\delta}\frac{V}{k+1}$. The result follows.

**Remark:** We prove that the maximum possible error of any flow in our top-$k$ algorithm is at most $\sqrt[\theta]{1-\delta}\frac{V}{k+1}$. Note that this is slightly larger than the worst-case error $\frac{V}{k+1}$ of Misra-Gries's algorithm. However, all flows in Misra-Gries's algorithm share the worst-case bound $\frac{V}{k+1}$. In contrast, our algorithm estimates the per-flow lower and upper bounds with three counters and the maximum error is $e_f$, which varies across flows. Our experiments show that our top-$k$ algorithm can achieve much smaller per-flow error bounds than Misra-Gries's algorithm in practice (§7.5). We leave the theoretical analysis of average-case errors to future work.