Toward Nearly-Zero-Error Sketching via Compressive Sensing

Qun Huang, *Peking University and Pengcheng Lab;* Siyuan Sheng, *Institute of Computing Technology, CAS;* Xiang Chen, *Peking University and Pengcheng Lab and Fuzhou University;* Yungang Bao, *Institute of Computing Technology, CAS;* Rui Zhang, Yanwei Xu, and Gong Zhang, *Huawei Theory Department*

https://www.usenix.org/conference/nsdi21/presentation/huang

This paper is included in the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Toward Nearly-Zero-Error Sketching via Compressive Sensing

Qun Huang[1,2] Siyuan Sheng[3] Xiang Chen[1,2,4] Yungang Bao[3] Rui Zhang[5] Yanwei Xu[5] Gong Zhang[5]
[1]Peking University [2]Pengcheng Lab [3]Institute of Computing Technology, CAS
[4]Fuzhou University [5]Huawei Theory Department

Abstract

Sketch algorithms have been extensively studied in the area of network measurement, given their limited resource usage and theoretically bounded errors. However, error bounds provided by existing algorithms remain too coarse-grained: in practice, only a small number of flows (e.g., heavy hitters) actually benefit from the bounds, while the remaining flows still suffer from serious errors. In this paper, we aim to design a nearly-zero-error sketch that achieves negligible per-flow error for almost all flows. We base our study on a technique named compressive sensing. We exploit compressive sensing in two aspects. First, we incorporate the near-perfect recovery of compressive sensing to boost sketch accuracy. Second, we leverage compressive sensing as a novel and uniform methodology to analyze various design choices of sketch algorithms. Guided by the analysis, we propose two sketch algorithms that seamlessly embrace compressive sensing to reach nearly zero errors. We implement our algorithms in OpenVSwitch and P4. Experimental results show that the two algorithms incur less than 0.1% per-flow error for more than 99.72% flows, while preserving the resource efficiency of sketch algorithms. The efficiency demonstrates the power of our new methodology for sketch analysis and design.

1 Introduction

Sketch algorithms have been widely adopted in flow-level monitoring. They maintain compact data structures that sacrifice a small portion of accuracy to be readily deployable in commodity network devices. Given their limited overheads and provable high accuracy, numerous sketch algorithms are designed to monitor various flow statistics, such as per-flow counting [49], heavy hitters [19, 25], denial-of-service victims [26,84] and traffic distributions [46]. These flow statistics form essential building blocks for network management.

Despite the sound theoretical bounds on the errors, existing sketch algorithms remain far from perfect for providing comprehensive guarantees for all flows. Ideally, it is expected to monitor *every* flow with *minimum* errors, which empowers various fine-grained network management operations such as responsive diagnosis [17, 51, 67] and precise failure localization [3, 50]. However, the bounds in existing algorithms are designed for specific traffic statistics such as heavy hitters or flow distributions. They are too coarse-grained when applied to all flows. As a result, *only a small portion of flows actually benefit from the provable error bounds.* For instance, for byte counting, many sketch algorithms guarantee an upper bound of per-flow error. Heavy hitters whose size is much larger than the bound can certainly achieve high accuracy as the maximum possible error is limited compared to their size. Nonetheless, such a bound is still unacceptable for most small flows that still suffer from poor accuracy.

In this paper, our goal is to explore nearly-zero-error (NZE) per-flow monitoring. We aim to achieve a negligibly small error (e.g., >99.99% flows are reported, and the estimated size of any reported flow has a <0.1% relative error compared to the true size). We base our study on a signal processing technique named *compressive sensing*. Our key insight is that: (1) compressive sensing provides near-perfect signal recovery with limited resources, which inspires us to apply it to flow monitoring; (2) compressive sensing is built on various matrix properties such as sparsity, which provides a powerful tool to study sketch algorithms, given that most sketch algorithms exhibit the same mathematical form as compressive sensing [21]. Even though some telemetry solutions also adopt compressive sensing [7, 16, 21, 35, 44, 83], our work addresses the design of NZE sketch, which is never studied.

In particular, we exploit compressive sensing in two lines. In the *first line*, we incorporate the near-perfect recovery technique of compressive sensing by regarding flow statistics as signals. However, our preliminary experiments show that it is non-trivial to adopt compressive sensing directly. This motivates the *second line* of our work that examines the suitability of compressive sensing for sketch algorithms and then designs new algorithms accordingly. Specifically, we leverage compressive sensing to propose a novel and uniform methodology to study sketch techniques: we formulate various sketch algorithms in forms of matrices and then quantitatively analyze their suitability to compressive sensing. Thus, instead of designing from scratch, we use the analysis results as a guideline for the algorithm design.

In summary, we not only propose new algorithms but also provide a new methodology to study sketch techniques from a perspective of compressive sensing. We make the following contributions:

- We investigate the feasibility of applying compressive sensing to flow monitoring. We evaluate two simple methods and show that simple utilization either suffers from poor scalability or fails to reach the expected accuracy level.
- We dissect existing sketch algorithms based on compres-

sive sensing theory. We formulate each sketch algorithm by inducing a matrix for it. We examine a fundamental matrix property namely *orthonormality* that ensures the correctness of compressive sensing. We find that induced matrices of existing algorithms fail to be orthonormal.

- We study the common approaches to build sketch algorithms from a perspective of matrix analysis. We analyze the impact of these approaches on the orthonormality of their induced matrices. We reveal the limitations of existing algorithms when combining with compressive sensing.
- We design two new algorithms that efficiently utilize the common approaches to embrace compressive sensing seamlessly. The two algorithms target suitability to compressive sensing to achieve nearly-zero errors, while prior algorithms provide only coarse-grained error bounds. Further, their design choices can be interpreted by matrix analysis, while existing algorithms are built on statistical analysis or empirical observations on hash conflicts. To our best knowledge, both the two aspects are never explored before.
- We implement our proposed algorithms atop both OpenVSwitch [62] and P4 [63]. Our evaluation results demonstrate that our algorithms achieve less than 0.1% relative error for more than 99.72% flows, while incurring zero false negatives and zero false positives, while consuming limited resources compared to state-of-the-art algorithms. We release our source code at **https://github.com/N2-Sys/NZE-Sketch**.

2 Problem

2.1 Sketch-based Flow Monitoring

We follow the line of approximate flow-level measurement [4, 35, 37, 49, 53, 54, 80, 82]. Flow-level monitoring defines a flow as a sequence of packets with the same *flow ID*, and computes its *flow values* based on the packet sequence. We focus on sketch algorithms that outperform sampling in accuracy [49] and hence have been extensively used in flow monitoring. A sketch algorithm records information of *every* packet in a *compact* data structure, so as to achieve high accuracy yet be readily deployed in commodity measurement points.

In a nutshell, a sketch algorithm comprises a collection of counters. It supports two operations: *update* and *query*. The update operation is performed in the data plane. For each packet, it selects several counters with hash functions and updates the selected counters to reflect the changes of flow values. The query operation is invoked by the control plane. The control plane periodically collects sketch structures from each measurement point, and performs the query operation to extract flow IDs and flow values from the structure.

Sketch algorithms allow a small but bounded accuracy drop to reduce resource overheads. Specifically, in a sketch algorithm, a counter is typically shared by multiple flows, which inevitably incurs some errors due to flow conflicts. Each sketch algorithm mitigates the errors with its specific algorithmic design. Backed by sound mathematical analysis, sketch algorithms usually provide theoretical bounds on the errors.

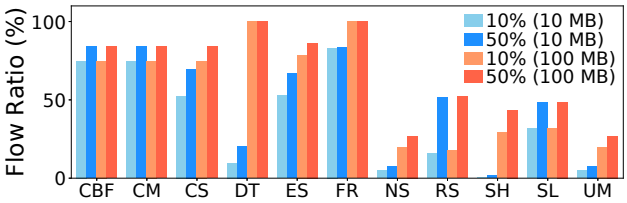


Figure 1: Fractions of flows that reach <10% and <50% per-flow errors in existing sketch algorithms.

2.2 Limitation

However, the theoretical guarantees provisioned by existing sketch algorithms are limited. Existing algorithms are typically designed to provide guarantees for specific flows (e.g., heavy hitters [8, 19, 25, 68] or super-spreaders [84]) and/or aggregated flow statistics (e.g., cardinality [28] or traffic distribution [46]). With regard to the specific scope, it is sufficient for a sketch algorithm to mitigate the overall hash conflicts only because bounding per-flow error is not a primary goal. Nonetheless, when extending an algorithm to the entire network traffic, the derived bounds are too coarse-grained to work for all flows. This leads to a considerable gap between theoretical analysis and practical results: only a small portion of flows actually benefit from the theoretical bounds, while the remaining flows still exhibit poor accuracy.

We consider an example of CountMin [20] to illustrate it. A CountMin sketch consists of r rows, each of which has w counters. When applying it to count per-flow bytes, it guarantees that the per-flow counting error is at most $\frac{2U}{w}$ with a high probability $1-\frac{1}{2^r}$, where U is the total byte count of all flows. Now we consider an interval with U =10 GB traffic, and configure $w=10^5$ and a sufficiently large r such that the probability $1-\frac{1}{2^r}$ is close to one. In this case, the error bound is around 210 KB. For extremely large flows, such a bound guarantees a small error (e.g., <2% relative error for a flow of 10 MB). However, the error is awfully huge for small flows whose byte counts are below the bound. Given the heavy-tailed traffic distribution, most flows are small. Thus, most flows suffer from low accuracy due to the loose bound.

We justify this observation via trace-driven experiments. We consider 11 sketch algorithms for per-flow packet counting: Counting Bloom Filter (CBF) [27], CountMin (CM) [20], CountSketch (CS) [15], Deltoid (DT) [19], ElasticSketch (ES) [80], FlowRadar (FR) [49], NitroSketch (NS) [53], RevSketch (RS) [68], SeqHash (SH) [8], SketchLearn (SL) [37], and UnivMon (UM) [54]. We use Caida [9] traces and partition our traces in two 2-second intervals where each interval contains 100 K flows. We employ two configurations for each algorithm: one with 10 MB memory that is around the maxi-

mum available memory in commodity switches [43, 56], and the other with 100 MB that indicates an ideal scenario that has plenty of memory resources. We set parameters as suggested in the original papers. Figure 1 presents the fractions of flows whose per-flow error is below 10% and 50%. With 10 MB, less than half flows can reach a per-flow error below 10% in most algorithms. The low accuracy is caused by the serious hash conflicts in these sketches. With 100 MB, the overall accuracy is improved. However, such huge memory consumption is not affordable in commodity switches.

3 Overview

Goals. We explore the methodology to design NZE sketch algorithms. Specifically, we expect that: (1) flow IDs are extracted with a negligible error probability (e.g., both false positive rate and false negative rate are below 0.01%), and (2) per-flow error is small (e.g., <0.1%) for almost all (e.g., >99%) flows. At the same time, we also aim to limit the resource usage such that the algorithms can be readily deployed.

The NZE monitoring forms the basis for various flow statistics, such as flow cardinality [28], super-spreaders and DDoS victims [30, 86], heavy hitters/changes [45], flow distributions [46], and entropy [34]. For each type of statistics, a lot of specific algorithms have been proposed. However, to our best knowledge, none of existing algorithms provide comprehensive and strict accuracy guarantees for all flows. Prior studies advocate that: (1) it is sufficient to address large flows, and (2) approximate monitoring is acceptable. Nevertheless, NZE monitoring for even small flows greatly benefits network management. For example, single-packet TCP flows typically indicate unsuccessful connection attempts, caused by DDoS attacks, service crashes, or software bugs. Without accurate monitoring for small flows, it is difficult to rapidly react to such events. On the other hand, NZE monitoring allows administrators to deal with the reported anomalies without concerns on false alarms or undetected events.

Key idea. Our study addresses three questions. (1) Is NZE monitoring theoretically feasible? (2) What are the key factors to achieve NZE monitoring? (3) How do the key factors be efficiently realized in practice?

To answer these questions, we base our work on compressive sensing [11–13, 24]. Compressive sensing is a signal processing technique that acquires high-dimensional signals with limited resources. Classical compressive sensing has two procedures. The *sensing procedure* records a signal by multiplying the signal with a matrix, while the *recovery procedure* reconstructs the signal with an optimization-based approach. We exploit compressive sensing in two aspects:

- We aim to incorporate the optimization-based recovery of compressive sensing to achieve near-zero errors. This is motivated by the sound theoretical guarantees provisioned by compressive sensing on its overheads and correctness. It has been demonstrated that the optimization-based approach can recover signals nearly perfectly in many areas such as image compression [11, 12]. Thus, similar results are expected if we regard flow values as signals.
- We also leverage compressive sensing to guide the design of NZE sketch. Here, compressive sensing serves as a general framework to study various sketch algorithms. In particular, compressive sensing exhibits the same mathematical form as sketch algorithms: prior works [18, 21, 55] show that sketch algorithms can be viewed as variants of compressive sensing. Even though each sketch algorithm exhibits its unique design that is quite different from classical compressive sensing, it can be formulated by a matrix (§4.3) and analyzed via matrix analysis.

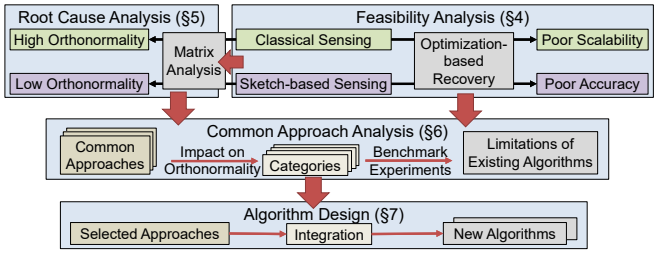


Figure 2: Workflow.

Assumptions. Our study makes two assumptions. First, network traffic is sparse. By sparsity, we mean that even though there are enormous possible flows (e.g., 2^{64} possible 2-tuple flows), the number of active flows is much smaller. This assumption has been justified in many measurement studies [5,66] and utilized in various recent works [35,83]. Second, we assume that a sketch algorithm contains a linear part in which each counter is updated linearly by a packet. Previous studies [18, 21] show that basic sketch algorithms, including CM, CS, and CBF, are linear structures; while we observe that many other sketches (e.g., UM, FR, ES, and SL) are built atop these basic sketches. We discuss how to handle the non-linear portion of a sketch in §4.3.

Workflow. Figure 2 outlines the workflow of our study.

- **Feasibility analysis (§4):** We investigate the feasibility of applying compressive sensing to flow monitoring. We consider two methods. The first method directly adopts classical compressive sensing, including its sensing and recovery procedures. The second method employs sketch algorithms to record per-packet information (referred to as *sketch-based sensing*). Then it formulates each sketch algorithm as a matrix and invokes the optimization-based recovery of classical compressive sensing. However, we find that the first method suffers from a scalability problem, while the second method has very low accuracy. This motivates us to dive into the fundamental theory of compressive sensing and design new algorithms.
- **Root cause analysis (§5):** We examine the root cause that leads to the poor accuracy when combining sketch-based

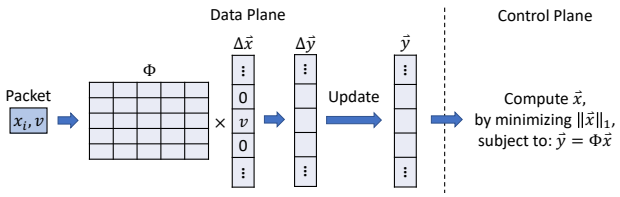


Figure 3: Utilization of compressive sensing for network monitoring.

sensing with the optimization-based recovery. Our study compares the matrices formulated by sketch algorithms with those in classical compressive sensing. We address a critical matrix property *orthonormality* (§1) that ensures the correctness of classical compressive sensing. Our benchmark experiments show that the matrices formulated by sketch algorithms lack sufficient orthonormality.

- **Common approach analysis (§6):** We identify common approaches to improve matrix orthonormality for sketch algorithms. Even though the approaches are also used in existing algorithms, we revisit these approaches in a novel methodology of matrix analysis. We group the approaches into four classes. For each class, we theoretically analyze the impact on matrix orthonormality. We also perform benchmark experiments to validate our analysis. Our analysis points out that the current utilization of these approaches is not efficient to combine with the optimization-based recovery of compressive sensing.
- **Algorithm design (§7):** We propose two algorithms that seamlessly combine sketch with the compressive sensing recovery. Based on the results of common approach analysis, we select appropriate approaches and efficiently integrate them to form the data plane of the two algorithms. Each design choice can be fully interpreted by matrix analysis. In the control plane, the two algorithms recover flow IDs and flow values by solving an optimization problem.

Discussion. Some recent studies have also applied compressive sensing in network measurement [7, 16, 21, 35, 44, 83]. However, they focus on recovering missing values in specific scenarios such as traffic matrices [16, 83] or network tomography [7]. In contrast, we utilize compressive sensing to (i) comprehensively dissect sketch algorithms, and (ii) guide the full design of NZE sketch.

Note that there are numerous variants of compressive sensing that reconstruct signals in different manners (e.g., LASSO [47] or using L_0 norm). In this paper, we focus on the original reconstruction approach that is based on matrix orthonormality [11, 12], given their simplicity and sound guarantees.

4 Feasibility Analysis

We introduce the fundamental concepts of compressive sensing in §4.1. Then we study two methods that apply compressive sensing to flow monitoring in §4.2 and §4.3, respectively.

4.1 Preliminary

Compressive sensing represents a signal as a *signal vector* $\vec{x}$ of length n. It includes two procedures to acquire $\vec{x}$.

Sensing procedure. The sensing procedure is responsible for recording $\vec{x}$ in a lightweight manner. Since the length n is usually a large number, compressive sensing linearly maps $\vec{x}$ into a *measurement vector* $\vec{y}$ of length m, where m is much smaller than n. Formally, the mapping can be represented as an $m \times n$ *sensing matrix* Φ, while $\vec{y}$ is computed as:

$$\vec{y} = \Phi \times \vec{x} \tag{1}$$

Recovery procedure. The recovery procedure is to reconstruct the signal vector $\vec{x}$ with Φ and $\vec{y}$. However, Equation (1) is an underdetermined system. It includes m linear equations for the n unknown variables in $\vec{x}$: the i-th element in $\vec{y}$ (denoted by y_i) and the i-row of Φ form a linear equation: $\sum_{j=1}^{n} \Phi_{i,j} \cdot x_j = y_i$. Since the number of variables n is much larger than the number of equations m, the number of possible solutions is infinite.

Compressive sensing addresses the underdetermined problem by introducing some prior knowledge. It assumes that $\vec{x}$ is sparse[1]. Then compressive sensing formulates an optimization problem:

$$\begin{aligned} \text{minimize:} \quad & \|\vec{x}\|_1, \\ \text{subject to:} \quad & \vec{y} = \Phi\vec{x} \end{aligned} \tag{2}$$

Here, compressive sensing chooses to minimize the L_1 norm of $\vec{x}$ because L_1 norm penalizes against the lack of sparsity [11, 12]. Therefore, a sparse vector satisfying Equation (1) is obtained. Theoretical analysis shows that the solution is close to the true $\vec{x}$ if some specific properties hold in Φ [11–13, 24]. We will study the properties in §5.

Utilization. Figure 3 depicts how to map the concepts of compressive sensing to those in network monitoring. Let n be the number of possible flows. For each type of flow statistic, all flow values form a vector $\vec{x}$ of length n. An element x_i indicates the value of the flow i. A measurement point maintains $\vec{y}$ in its memory. For each packet, it identifies the flowkey i such that the packet can be considered as a change to $\vec{x}$ denoted by $\Delta\vec{x}$. The measurement point multiplies $\Delta\vec{x}$ with the matrix Φ to form the update to $\vec{y}$ (denoted by $\Delta\vec{y}$). Then it applies the update to $\vec{y}$. The control plane collects $\vec{y}$ and invokes the optimization-based recovery in Equation (2) to reconstruct $\vec{x}$. Note that we do not need to explicitly maintain Φ, $\vec{x}$, $\Delta\vec{x}$, and $\Delta\vec{y}$ in memory. Instead, we compute their elements on demand. For example, we compute an element $\Phi_{k,i}$ when we update the k-th counter in $\vec{y}$ with flow indexed by i.

In practice, network administrators can directly utilize a classical matrix Φ [12, 75]. They can also propose their own method and formulate it in the form of Equation (1). We study

[1] For non-sparse $\vec{x}$, it needs to be transformed to another sparse vector first. We omit this case because network traffic exhibits high sparsity (§3).

the classical methods in §4.2 and formulate sketch algorithms using compressive sensing in §4.3.

4.2 Method 1: Classical Sensing

Accuracy. We first consider a method that direct utilizes classical sensing matrix Φ. We evaluate the accuracy of the classical sensing method via experiments with the same setup as that in §2.2. We employ four types of commonly used sensing matrices Φ: (1) Gaussian Matrix (GM) [75], (2) Bernoulli Matrix (BM) [12], (3) Incoherence Matrix (IM) [12], and (4) Fourier Matrix (FM) [12]. To reconstruct $\vec{x}$, we leverage two algorithms: the L_1 minimization approach that solves the optimization problem with the simplex method [22], and a greedy algorithm named Orthogonal Matching Pursuit (OMP) [64]. The four types of sensing matrices and two recovery algorithms produce eight approaches in total. The results show that all the eight approaches can recover flow IDs and flow values perfectly: zero false positives, zero false negatives, and zero per-flow error. Our results show that 400 KB memory is sufficient to achieve perfect recovery, which is much smaller than sketch algorithms (§2.2). The detailed accuracy trend with different memory settings is in Table 3 in Appendix.

Scalability problem. However, the classical sensing method suffers from a scalability problem. The classical sensing matrices are dense matrices in which all elements are non-zero. Thus, each packet needs to update m (above 10^4) counters in $\vec{y}$. This is infeasible for commodity devices. In software switches, updating so many counters fails to keep pace with packet streams with the slow CPUs. In hardware switches, the updates far exceed the available computational units. Thus, classical sensing can only accommodate limited flows.

Note that the scalability problem does not occur in other compressive sensing applications in which $\vec{x}$ does not vary (e.g., image compression). In those scenarios, $\vec{y}$ is computed only once using the constant $\vec{x}$.

4.3 Method 2: Sketch-based Sensing

Matrix formulation. Sketch algorithms incur limited per-packet operations, which addresses the scalability problem in §4.2. we follow prior studies [18, 21] that regard sketch as *linear* mapping and formulate it in the form of $\vec{y} = \Phi\vec{x}$. Let m be the number of linear counters. For $\vec{y}$, we index the m counters and stack them as a vector $\vec{y}$ of length m. For Φ, we form Φ with m rows and n columns, where each column represents a flow while each row represents one counter in $\vec{y}$. Each element $\Phi_{i,j}$ implies that the counter y_i is incremented by $\Phi_{i,j}$ if the value x_j of flow j changes by one.

Examples. We present an example of CountMin in Figure 4. We consider four flows, i.e., $\vec{x}$ has its length n=4. We employ two rows and configure three counters in each row in the sketch. Hence, $\vec{y}$ has length $m = 6$. In each row, a packet (k, v)

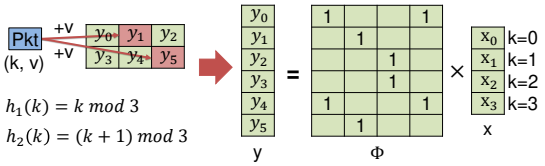


Figure 4: Matrix formulation of CountMin.

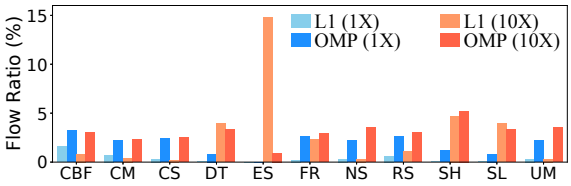


Figure 5: Ratio of flows that reach <50% per-flow errors in sketch-based sensing.

selects one counter with a hash function and increments it by v. Thus, we have $\Phi_{i,k} = 1$ if flow k is hashed to counter i, and $\Phi_{i,k} = 0$ otherwise. For each more flow, we may add a column to Φ and derive the column elements in the same method. We present more examples in Appendix.

Nonlinear structures. Not all components in a sketch are linear mapping. These components cannot be formulated by matrices. For example, FlowRadar maintains a set of counters that encode flow IDs via XOR operations [49]. We do not incorporate such nonlinear components in the optimization problem when reconstructing $\vec{x}$. Instead, we employ them to verify the correctness of the reconstructed $\vec{x}$. Specifically, we recompute these nonlinear structures with the reconstructed $\vec{x}$ and compare them with the original ones. For example, in FlowRadar, we encode all recovered flow IDs via the same XOR operations. We compare the new encoded results with the original XOR results to validate the correctness.

Results. We evaluate the sketch-based sensing method. We consider the sketch algorithms in §2.2. For each algorithm, we perform both L_1 minimization and OMP for the recovery. We employ two memory configurations: one with the same amount of memory as classical sensing, and the other with $10\times$ memory. Figure 5 presents the ratio of flows whose error is below 50%. We see that even with $10\times$ memory, all algorithms suffer from extremely low accuracy when using the optimization-based recovery. The results are even worse than those using their original query operations (see §2.2). The reason is that the matrices derived from existing sketch algorithms do not exhibit the required properties of compressive sensing although they exhibit the same form (see 5). This suggests us to explore new methods to boost sketch-based sensing to embrace compressive sensing, provided by the extensive study on the accuracy of compressive sensing [11–13, 24].

5 Root Causes

We examine the root cause of the poor accuracy in §4.3. Our methodology is to examine whether key properties of classical

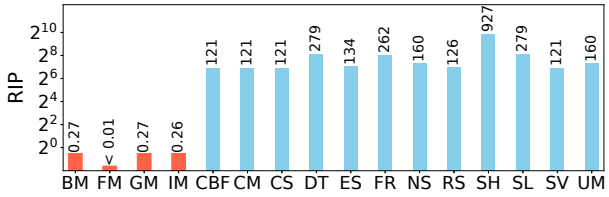


Figure 6: RIP of classical and sketch-based sensing.

sensing hold in sketch-based sensing.

Key properties. Compressive sensing guarantees its correctness by two properties: the *sparsity* of $\vec{x}$ and the *orthonormality* of Φ. Specifically, any orthonormal matrix Φ preserves the norms (and hence differences) for sparse vectors: given arbitrary two different sparse vectors $\vec{x}_1$ and $\vec{x}_2$, their mappings under matrix Φ (i.e., $\Phi\vec{x}_1$ and $\Phi\vec{x}_2$) remain distinct. Thus, when a sparse vector $\vec{x}^*$ that satisfies $\Phi\vec{x}^* = \vec{y}$ is found, it is must be equal to the desired $\vec{x}$ (otherwise two different vectors $\vec{x}^*$ and $\vec{x}$ have the same mapping, which compromises the property of the orthonormal matrix Φ) [12]. Since $\vec{x}$ is already sparse (§3), we only address whether Φ is orthonormal.

Orthonormal matrix and RIP. Unfortunately, orthonormality cannot hold in Φ, because an orthonormal matrix is requires to be a square matrix, but the number of rows is smaller than the number of columns in Φ. Compressive sensing deals with this issue with a notion of *restricted isometry property (RIP)* [10], which serves as an approximation of being fully orthonormal. RIP characterizes the extent to which Φ preserves the norm of sparse signals.

At a high level, for any sparse vector $\vec{x}$, $\Phi\vec{x}$ is its mapping under the matrix Φ. If Φ is highly orthonormal, the norm of $\Phi\vec{x}$ (denoted by $\|\Phi\vec{x}\|_2$) must be close to that of $\vec{x}$ (denoted by $\|\vec{x}\|_2$). Therefore, RIP evaluates the difference between $\|\vec{x}\|_2$ and $\|\Phi\vec{x}\|_2$. Since Φ should work for an arbitrary sparse vector $\vec{x}$, RIP is calculated as a sequence of *isometry constants* $\{\delta_S\}$. Each δ_S in the sequence is the maximum relative difference between the norms of $\Phi\vec{x}$ and $\vec{x}$ among all S-sparse signals:

$$\delta_S = \sup\{\frac{|\,\|\Phi\vec{x}\|_2 - \|\vec{x}\|_2\,|}{\|\vec{x}\|_2} \text{ for any S-Sparse } \vec{x}\} \qquad (3)$$

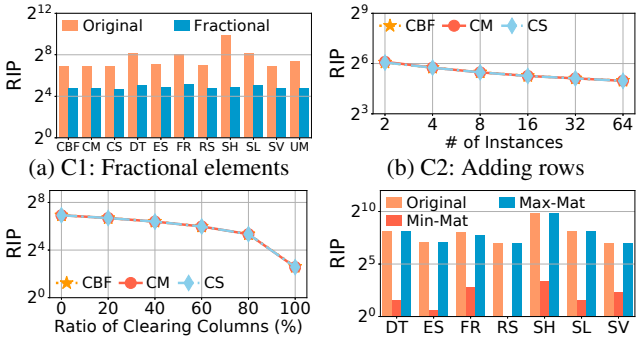
Benchmark results. We measure the RIP of both classical sensing matrices (§4.2) and the matrices induced by sketch algorithms (§4.3). We present RIP as δ_S, the isometry constant for S-sparse vectors, where S is the number of actual flows in each interval. Figure 6 shows that classical sensing matrices have RIP below 0.3. In contrast, RIP is above 120 in all sketch-induced matrices. The large RIPs degrade the efficiency of compressive sensing reconstruction.

6 Common Approach Analysis

We examine common approaches in general sketch design. Based on their impacts on matrix orthonormality, we categorize the approaches into four classes. For each class, we analyze its matrix property and use RIP as the metric to quantify the effectiveness. Although some approaches have been used in existing sketch algorithms (see Table 1), we study them from novel perspectives. First, we target the suitability of these approaches to compressive sensing, while existing algorithms study them for specific purposes. Second, we quantify the efficiency of these approaches via matrix analysis, while previous algorithms address probabilistic error bounds.

Algorithm	C1	C2	C3	C4
CU Sketch [25]	Conservative update			
Deltoid [19]		Multiple CM instances		Flow extraction
ElasticSketch [80]				Traffic splitting
FlowRadar [49]		Multiple Bloom Filters	Bloom Filter	Flow extraction
NitroSketch [53]	Sampling	Multiple CS instances	Heap	
RevSketch [68]				Flow extraction
SeqHash [8]		Multiple CM instances		Flow extraction
SketchLearn [37]		Multiple CM instances		Flow extraction
SketchVisor [35]				Traffic splitting
UnivMon [54]		Multiple CS instances	Heap	
SeqSketch	Fractional update		Bloom Filter + Controller	Splitting + Controller
EmbedSketch	Fractional update		Bloom Filter + controller	Extraction + Controller

Table 1: Common approaches in sketch algorithms.



(a) C1: Fractional elements (b) C2: Adding rows

(c) C3: Clearing columns (c) C4: Matrix decomposition

Figure 7: Impact of common approaches.

6.1 Class 1 (C1): Fractional Elements

Matrix analysis. We observe that elements in sketch-based sensing matrices are integers, which leads to the norm of matrix columns above one. However, an orthonormal matrix requires column vectors with norm one. Thus, the first class is to employ fractional matrix elements whose values are less than one, such as to reduce the norm of each column.

Benchmark results. Figure 7(a) evaluates the impact of fractional elements in existing sketch algorithms. For a sketch, we replace each element with a randomized value $1/\sqrt{t}+\sigma$, where σ is sampled from a Gaussian distribution with its mean equal to zero. Thus the mean of elements in the ma-

trix is $1/\sqrt{t}$. Here, t is the number of counters accessed by a packet. Thus, the expected norm of each column vector is one. We see that RIP is decreased by 40% in all cases.

Approaches. In existing algorithms, there are two approaches producing fractional elements.

- **Sampling:** Sampling techniques [69, 70] discard some packets. For each flow, only partial packets contribute to $\vec{y}$. Thus, the elements in Φ are less than one. NitroSketch [53] has combined an adaptive sampling in its design.
- **Conservative updates:** In general, a packet incurs several updates in a sketch algorithm. Conservative update [25] preserves only the smallest update and drops the others. This also leads to smaller elements in Φ because not all updates are included.

Limitation. However, sampling and conservative updates are hard to be formulated as matrices. To obtain the exact fractional elements in Φ, it needs to track exact per-flow packet loss or update drops, which inevitably incurs excessive overheads and cancels out the benefit of sketch.

6.2 Class 2 (C2): Adding Rows

Matrix analysis. The second class is to add more rows to the matrix Φ. Ideally, two columns are orthonormal if and only if their nonzero elements occur in different positions. This implies that the two flows have no conflicts in all counters. Since each counter contributes one row in Φ (§4.3), adding rows means to configure more counters to reduce flow conflicts. Hence, column vectors become more orthonormal.

Approaches. In addition to simply allocating more counters to a single sketch, a common approach is to use multiple instances of sketch structures. For instance, FlowRadar [49] contains two Bloom Filters; UnivMon [54] employs multiple CS instances and filters flows for each instance; Deltoid [19] and SketchLearn [37] maintain multiple CM instances while each instance is updated based on the bits of flow IDs.

Benchmark results. Figure 7(b) shows RIP with respect to various number of sketch instances. We consider three commonly used basic sketch algorithms: CM, CS, and CBF. We see that the RIP decreases as the number of instances grows. With 64 instances, RIP is reduced by nearly 75%.

Limitation. However, the resulting RIP is still much higher than that of classical sensing matrices (§5). Although we can further reduce RIP with more instances, adding instances consumes more memory. It also incurs excessive usage of computational resources to update multiple instances.

6.3 Class 3 (C3): Clearing Columns

Matrix analysis. The third class is to clear elements of some columns. Recall that a column indicates the contribution of an unknown variable (§4.3). Clearing one column means to exclude a variable, which simplifies the optimization problem and hence improves accuracy.

Approaches. Identifying columns that can be cleared is equivalent to detecting flow IDs that never occur, such that discarding the flows does not compromise the results. Two approaches can track flow IDs in existing algorithms.

- **Heap:** CountMin [20], UnivMon [54] and NitroSketch [53] use a heap to store flow IDs whose flow values satisfy specific conditions (e.g., above a pre-defined threshold).
- **Bloom Filter:** Bloom Filter records Flow IDs compactly with bit arrays. An example is FlowRadar [49] that uses Bloom Filter to avoid duplicate flow IDs.

Benchmark results. Figure 7(c) measures how RIP varies as the ratio of cleared useless columns. Due to the interest of space, we present three sketches here and put the remaining results in Appendix. With no columns cleared, RIP is above 120 for all the three sketch algorithms. RIP significantly decreases as the number of cleared columns grows. It becomes 6 when all useless columns are cleared.

Limitation. However, tracking all Flow IDs with existing approaches is bounded by resource restrictions in switches. For the heap-based approach, per-flow tracking is infeasible due to the memory usage of heap. For BF, since it only examines the occurrence of flow IDs, extra resources are needed to store flow IDs (e.g., XOR arrays in FlowRadar [49]). For the controller-based approach, it needs careful design to avoid bandwidth exhaustion.

6.4 Class 4 (C4): Matrix Decomposition

Matrix analysis. The final class decomposes Φ as the sum of several component matrices. The decomposition distributes non-zero elements in Φ into different components. Thus, their conflicts are alleviated and hence each component becomes more likely to be orthonormal.

Approaches. There are two possible approaches to distribute flows and hence decompose matrix Φ.

- **Traffic splitting:** Traffic splitting employs multiple algorithmic parts in the data plane and splits traffic into different parts. Each part can produce a component matrix individually. For example, SketchVisor [35] maintains a fast path and a normal path, and directs traffic into either path based on real-time workloads. ElasticSketch [80] consists of a heavy part and a light part: traffic that is evicted from the heavy part enters the light part.
- **Flow extraction:** We can also form a component matrix by extracting flows from the sketch structure. These algorithms usually embed specific features in the sketch structures for the extraction. For example, FlowRadar [49] estimates the number of distinct flows in each counter (i.e., a row in Φ) and iteratively extracts from the counters with exactly one flow. Deltoid [19] and SketchLearn [37] extract from rows with large corresponding flow values.

Benchmark results. Figure 7(d) compares RIP before and after the decomposition. We present the component matrices with the minimum RIP (Min-Mat) and maximum RIP (Max-Mat). We observe that the orthonormality is significantly improved in each Min-Mat. For example, the original RIP of DT is nearly 1000, but it is reduced to 2.92 in Min-Mat.

Limitation. However, Max-Mat still exhibits high RIP in all algorithms. For some algorithms, the RIP of Max-Mat is close to that in the original matrix. We find that the decomposed traffic is limited because it needs extra structures to split traffic or extract flows. When resources are bounded, limited traffic can be decomposed.

7 New Algorithms

Motivation. §6 points out that existing algorithms fail to produce highly orthonormal matrices. The key issue is that they are not tailored for compressive sensing. On the one hand, the four classes C1-C4 are not realized efficiently. On the other hand, they do not collectively combine the approaches for compressive sensing. Thus, we need new algorithms that realize and combine the common approaches more efficiently.

Design choices. To better embrace compressive sensing, we examine each class in §6 to employ appropriate approaches to combine with compressive sensing.

- **C**1: As both sampling and conservative updates are hard to formulate, we realize a novel method of fractional updates. Specifically, for a packet (k,v), we use an additional hash function $g(\cdot)$ to change its value from v to $v \cdot g(k)$. Here, $g(\cdot)$ generates a value with its mean equal to $\frac{1}{\sqrt{r}}$ where r is the number of rows in the sketch. Thus, the expected norm of column vectors is reduced to one.
- **C**2: We discard C2 because of its excessive resource usage.
- **C**3: We employ a control-based approach to enhance Bloom Filter. Specifically, we store flow IDs in the controller. Since the controller has enough memory, the flow IDs can be recorded with zero errors. To reduce bandwidth usage, we employ a Bloom Filter to eliminate duplicate transfers.
- **C**4: We separate large flows as key-value pairs and small flows in the sketch with fractional values (referred to as fractional sketch). It has been proved that such separation can be realized with limited overheads [35,71,80]. To make *each* decomposed component matrix highly orthonormal, we also leverage the controller to steer the traffic in key-value pairs and the fractional sketch.

In summary, we maintain three types of components in the data plane: (1) key-value pairs to track large flows, (2) fractional-valued sketch to record small flows and (3) a Bloom Filter. We propose two algorithms that combine them in different manners. The first algorithm SeqSketch arranges the components sequentially, while the second algorithm EmbedSketch embeds the key-value table and Bloom Filter into the sketch arrays such that key-value pairs can be extracted from sketch buckets. SeqSketch consumes less memory, while EmbedSketch needs fewer computational units. Network administrators can select the more suitable algorithm based on their resource budget.

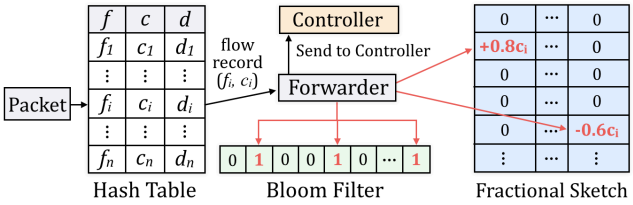


Figure 8: Structure of SeqSketch.

Algorithm 1 SeqSketch Data Plane

Input: Packet (k,v)

```
1:  procedure UPDATE(k, v)
2:      j = hash(k)
3:      if H[j] is ∅ then
4:          H[j].f = k, H[j].c = v, and H[j].d = 0
5:      else if H[j].f == k then
6:          H[j].c = H[j].c + v
7:      else
8:          H[j].d = H[j].d + v
9:          if H[j].d > H[j].c then
10:             Send (H[j].f, H[j].c) to controller
11:             H[j].f = k, H[j].c = v, and H[j].d = 0
12:         else
13:             for all row i in FS do
14:                 Compute j = h_i(k)
15:                 Increment counter (i, j) by g_i(k) · v
16:             if k ∉ BF then
17:                 Send k to controller
18:                 Insert k to BF
```

7.1 SeqSketch

Data structure. Figure 8 presents an overview of SeqSketch. SeqSketch organizes its key-value pairs in a hash table H, and employs a *forwarder* to connect the hash table, Bloom Filter BF and fractional sketch FS. Every packet first enters the hash table H. Each tuple in H has three fields to identify large flows: apart from flow ID f, two counters c and d record flow values belonging and not belonging to f, respectively. The hash table evicts records of potential small flows based on the two counters when conflicts occur. The forwarder transfers an evicted record if it is a new flow, or sends it to the fractional sketch. It uses the Bloom Filter to record all occurred flows and examine new flows. The eviction is cheap because it incurs limited operations for each evicted record. Given the heavy-tailed distribution of network traffic, the hash table H absorbs a large portion of traffic, which alleviates the memory usage of BF and FS. Thus, SeqSketch is memory efficient.

Data plane. Algorithm 1 outlines how SeqSketch processes

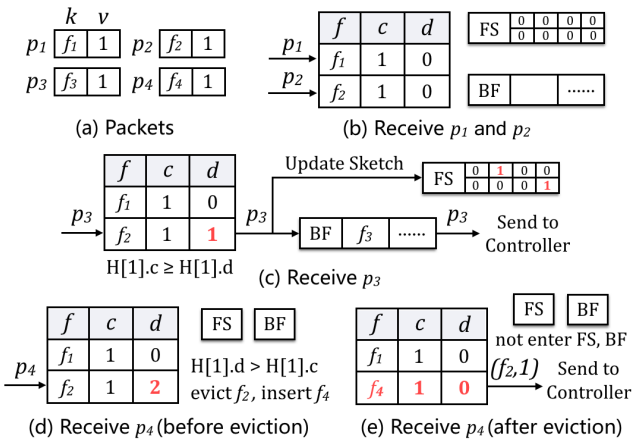


Figure 9: Example of SeqSketch.

a packet. When a packet (k,v) arrives, we first compute its position in the hash table H (Line 2). If its hashed entry is empty, a new record is created for this packet (Lines 3-4). If the entry already exists, there are two cases. First, if the existing entry has the same ID as the packet, the counter c is incremented (Lines 5-6). Second, if f and k are different, we add d by v (Line 8). At the same time, we need to evict the flow record of f or the packet (k,v) (Lines 9-18). When d is larger than c, we send the record to the controller (Line 10), and insert a flow record for k (Line 11). Otherwise, we evict (k,v) to FS (Line 13-15). In this case, the forwarder queries its Bloom Filter to examine whether the flow ID appears before. If it is a new flow, the ID is also forwarded to the controller (Line 16-18). Note that we increment each counter in FS with a fractional value instead of v (Lines 14-15).

Example. Figure 9 presents an example with two buckets in H. H directly inserts the first two packets p_1 and p_2 (Figure 9(b)). For the third packet p_3, H maps it to $H[1]$ that already records another flow f_2 . To deal with the conflict, H increments $H[1].d$. Since $H[1].d$ does not exceed $H[1].c$, p_3 is delivered to FS and BF. FS updates its counters with p_3, while BF transfers its flow ID to the controller because it is a new flow (Figure 9(c)). Finally, p_4 enters H and is also hashed to $H[1]$. Since p_4 does not belong to f_2, H increments $H[1].d$ by one. Since $H[1].d$ exceeds $H[1].c$ (Figure 9(d)), we evict $H[1]$ and insert p_4 (Figure 9(e)).

Control plane. We recover flow IDs and flow values by formulating an optimization problem. There are three portions of traffic: that in the hash table H, that transferred to the controller, and that in FS. Denote flow values in the three portions by $\vec{x}_H$, $\vec{x}_C$ and $\vec{x}_S$, respectively. Since $\vec{x}_H$, $\vec{x}_C$ can be obtained directly, we only need to solve $\vec{x}_S$ by formulating the per-flow update of FS as Φ and its counters as $\vec{y}$:

$$\begin{aligned} \text{minimize:} \quad & \|\vec{x}_S\|_1, \\ \text{subject to:} \quad & \vec{y} = \Phi \vec{x}_S \end{aligned} \tag{4}$$

Algorithm 2 EmbedSketch Data Plane

Input: Packet (k,v)

1: **function** UPDATEBUCKET(k, v, i, j)
2: $V_{i,j} = V_{i,j} + g_i(k)$
3: **if** $f_{i,j}$ is empty **then**
4: $f_{i,j} = k$, $c_{i,j} = v$, $d_{i,j} = 0$
5: **else if** $f_{i,j}$ is k **then**
6: $c_{i,j} = c_{i,j} + v$
7: **else**
8: $d_{i,j} = d_{i,j} + v$
9: **if** $d_{i,j} > c_{i,j}$ **then**
10: Send $(f_{i,j}, c_{i,j})$ to controller
11: $f_{i,j} = k$, $c_{i,j} = v$, $d_{i,j} = 0$
12: **else**
13: **if** $k \notin BF_{i,j}$ **then**
14: Send k to controller
15: Insert k to $BF_{i,j}$
16:
17: **procedure** UPDATE(k,v)
18: **for** row $i = 1,2,...,r$ **do**
19: $j = h_i(k)$
20: UPDATEBUCKET(k, v, i, j)

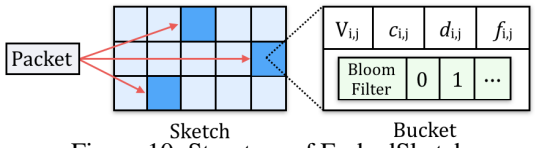


Figure 10: Structure of EmbedSketch.

7.2 EmbedSketch

Data structure. Figure 10 depicts EmbedSketch. It maintains a sketch with r rows. Each row i has two hash functions (h_i to select counters and g_i to generate fractional values) and w buckets. A bucket (i,j) in EmbedSketch consists of: (i) a counter $V_{i,j}$, which denotes the total values hashed to this bucket, (ii) $f_{i,j}$, which denotes the flow ID of the candidate for the largest flow in the bucket, (iii) $c_{i,j}$, which denotes the aggregated value of f, (iv) $d_{i,j}$, which denotes the total value of other flows in the bucket, and (v) a Bloom Filter $B_{i,j}$ that records flow IDs in this bucket. Essentially, EmbedSketch distributes monitoring operations in its buckets. This mitigates per-bucket hash conflicts. Thus, one hash function in each bucket is sufficient (see §7.4).

Data plane. Algorithm 2 details how EmbedSketch processes a packet (k,v). For each row i, EmbedSketch computes a bucket with $h_i(k)$ and updates the bucket (Lines 18-20). To update a bucket (i,j), EmbedSketch first increments $V_{i,j}$ by $g_i(k) \cdot v$ (Line 2). If the existing candidate $f_{i,j}$ equals to k, $c_{i,j}$ is also incremented (Lines 5-6). Otherwise, EmbedSketch increments $d_{i,j}$ (Line 8) and determines to evict either $f_{i,j}$ (Lines 9-11) or k (Lines 13-15). If $f_{i,j}$ is evicted, a record $(f_{i,j}, c_{i,j})$ is transferred to the controller (Line 10). At the

same time, EmbedSketch uses k as the new candidate and sets $c_{i,j} = v$ and $d_{i,j} = 0$ (Line 11). Otherwise, if k is evicted, EmbedSketch queries its local Bloom Filter $B_{i,j}$ (Line 13). If k is a new ID, EmbedSketch forwards it to the controller and updates $B_{i,j}$ to include k (Lines 14-15).

Control plane. We form $\vec{y}$ with all $r \times w$ counters of $V_{i,j}$ in EmbedSketch. Traffic in $V_{i,j}$ comprises three portions: (1) each $f_{i,j}$ contains its value $c_{i,j}$, (2) per-flow values transferred to the controller, and the remaining traffic in $V_{i,j}$. Thus, we denote per-flow values in the three portions by $\vec{x}_f$, $\vec{x}_C$ and $\vec{x}_R$, respectively. Since only $\vec{x}_R$ is unknown, we build the following optimization problem:

$$\begin{aligned} \text{minimize:} \quad & \|\vec{x}_R\|_1, \\ \text{subject to:} \quad & \vec{y} = \Phi(\vec{x}_f + \vec{x}_C + \vec{x}_R) \end{aligned} \tag{5}$$

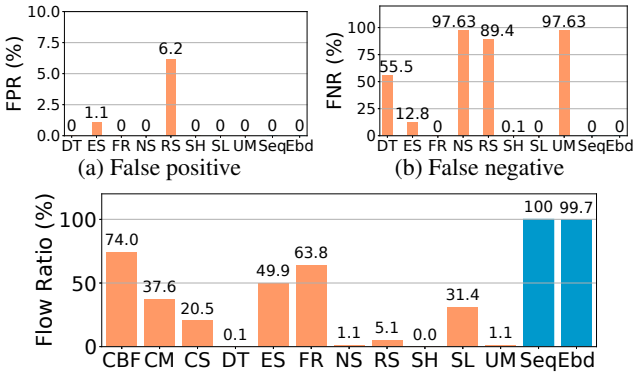
7.3 Parameters

We need to configure the three components: the fractional sketch (FS), the Bloom Filter (BF), and key-value pairs (KV).

Fractional sketch. For FS, two rows are sufficient as our optimization-based recovery does not need many rows to alleviate hash conflicts. However, compressive sensing theory requires a minimum amount of counters: $C \cdot S\log_2(n/S)$ (c.f. Equation(13) in [13]), where n is the number of possible flows, S is the expected number of actual flows, and C is a small positive number. In practice, we can select a proper C to make memory usage fits the device. For example, to monitor around S=100K 2-tuple flows ($n = 2^{64}$), setting C=0.1 leads to around 472K counters. If we employ 32-bit counters, the total memory of FS is 1888 KB.

Bloom Filter. The Bloom Filter BF determines the accuracy of the received flow IDs in the control plane. A false flow ID indicates 100% relative error for that flow, which seriously compromises the recovery accuracy. Thus, we need to carefully configure BF. The size of BF depends on the expected number of flows S. According to [18] (c.f. §5.2.5), the false positive rate of Bloom Filter is $(0.6185)^{m/S}$ where m is the length of Bloom Filter, if we set the number of hash functions to its optimal value $\frac{m}{S}\ln 2$. In our case, a false positive in Bloom Filter means wrongly clearing a column in Φ. To achieve our goal of $< 1\%$ error probability for flow ID extraction, we need to bound the false positive rate of the Bloom Filter below 1%. This requires $m = 9.6S$. For $S = 100$ K flows, this leads to a Bloom Filter with 120 KB. For SeqSketch, we employ the optimal number of hash functions: $9.6\ln 2 \approx 7$. For EmbedSketch, since the Bloom Filter is distributed across buckets, one hash suffices to achieve low error probability.

Key-value pairs. In EmbedSketch, each bucket maintains one key-value pair by design. For SeqSketch, it can employ the same amount for simplicity.



(a) False positive (b) False negative

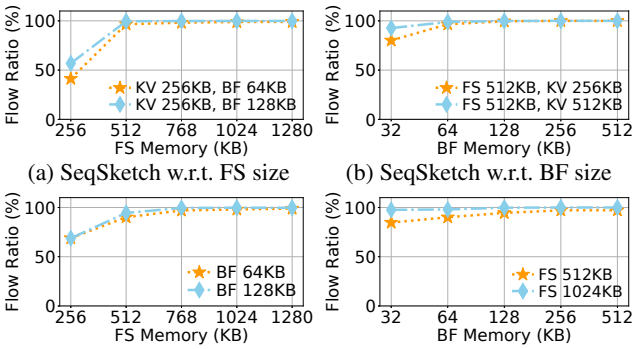
(c) Fractions of flows with relative errors less than 0.1%

Figure 11: (Experiment 1) Accuracy.

7.4 Evaluation

Setup. We implement both software version and hardware version of the two algorithms (see Appendix). We evaluate them via trace-driven experiments. We use the CAIDA-2018 backbone trace [9] and two data center traces [5]. We present 2-tuple flows and count their packets, while other flow definitions (e.g., 5-tuple) and statistics have similar results. We partition each trace into equal-length intervals. Due to the interest of space, we present the results with 2-second intervals, each of which has around 100 K flows. More results are in Appendix. We present the average results across all intervals. Here, we omit the standard deviations because the standard deviations are negligible. When measuring accuracy (Experiments 1 and 2), we run both update and query operations in a server with 36 CPU cores (2.6GHz each) and 128 GB memory to process the traces. When measuring resource overheads (Experiments 3 to 6), we build a testbed with 16 servers and a Barefoot Tofino switch [79]. Each server has a 40Gbps NIC for traffic transfers and a 10Gbps NIC to connect to the controller. We deploy our algorithms in the switch. Each server replays our traces and evenly sends the traces to others. We follow §7.3 to configure key-value pairs (KV), the Bloom Filter BF, and the fractional sketch FS.

Experiment 1: Accuracy (Figure 11). We compare the accuracy of SeqSketch (Seq) and EmbedSketch (Ebd) with 11 sketch algorithms. Every algorithm has its suggested theoretical configuration (see Appendix). However, they fail to achieve NZE even when we allocate 100 MB memory (§2.2). Thus, for a fair comparison, each algorithm is allocated with the same amount of memory as ours for a stress test. In Figure 11(a) and Figure 11(b), we exclude CBF, CM, and CS because they cannot extract flow IDs by design. We find that more than half existing algorithms have nearly zero false positive rate and false negative rate. However, none of existing algorithms achieve high accuracy for all flows. Recall §2.2, even when we allocate more memory (10 MB) and relax the desired per-flow error to 50%, the ratio of accurate flows in existing algorithms is less than 85%. In contrast, EmbedS-



(a) SeqSketch w.r.t. FS size (b) SeqSketch w.r.t. BF size

(c) EmbedSketch w.r.t. FS size (d) EmbedSketch w.r.t. BF size

Figure 12: (Experiment 2) Robustness to various memory configurations.

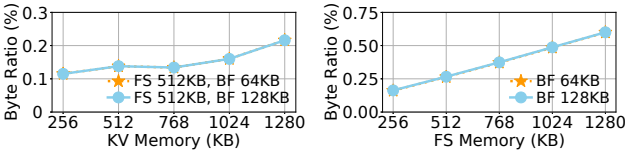
Name	PHV (Bytes)	VLIW	ALU	Stage
ElasticSketch	163 (21.22%)	13 (3.39%)	9 (18.75%)	10 (83.33%)
FlowRadar	134 (21.22%)	11 (2.86%)	15 (31.25%)	10 (83.33%)
SketchLearn	156 (20.31%)	11 (2.86%)	33 (68.75%)	8 (83.33%)
UnivMon	132 (17.19%)	13 (3.39%)	33 (68.75%)	12 (100%)
SeqSketch	151 (19.66%)	12 (3.12%)	13 (27.08%)	8 (66.67%)
EmbedSketch	137 (17.84%)	10 (2.60%)	6 (12.50%)	8 (66.67%)

Table 2: (Experiment 3) Switch resource usage.

ketch bounds the error below 0.1% for more than 99.72% and SeqSketch covers all flows. The reason is that existing algorithms depend on a large amount of memory to fully resolve hash conflicts. However, our algorithms recover flow values by solving an optimization problem based on compressive sensing, which is not so sensitive to hash conflicts.

Experiment 2: Robustness (Figure 12). We measure the ratio of flows with an error less than 0.1% in different configurations. To study the accuracy as memory changes, we fix two components and vary the size of the remaining one. For SeqSketch in Figure 12(a), when FS has 256KB, only 68% flows reach the accuracy level because the memory is far smaller than a reasonable size. However, the ratio increases to nearly 100% as the sketch size increases. Figure 12(b) shows that the accuracy remains stable for different *BF* configurations. Even with 32 KB (25% of the expected memory as §7.3), more than 96% flows remain per-flow error below 0.1%. In EmbedSketch, since each bucket of *FS* embeds one *KV* pair, we either fix *BF* and vary *FS* (Figure 12(c)) or vice versa (Figure 12(d)). We observe similar trends: the accuracy is low with 256 KB *FS* but grows as the size of *FS*, while remaining stable for various *BF* size.

We also find that SeqSketch is more memory efficient than EmbedSketch. For SeqSketch, 832 KB memory (512 KB *FS*, 64 KB *BF*, and 256 KB *KV*) is sufficient to achieve near-zero error. In contrast, EmbedSketch requires around 2.5 MB memory to reach the same level of accuracy, including at least 512 KB *FS*, 64 KB *BF* and 2048 KB *KV*. Here, the 2048 KB *KV* comes from the per-bucket key-value pairs. Recall that



(a) SeqSketch (b) EmbedSketch

Figure 13: (Experiment 4) Bandwidth usage.

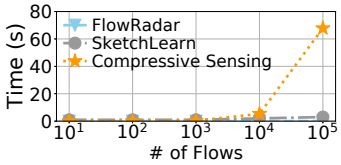


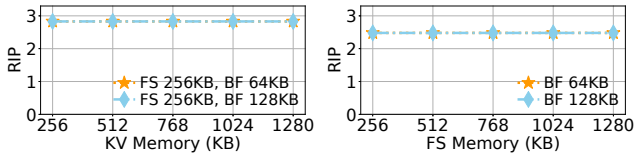
Figure 14: (Experiment 5) Recovery time.

each key-value pair occupies 16 bytes, which is 4× of a *FS* counter. Thus, *KV* consumes as 4× memory as *FS*. The root cause for different memory usage is that in SeqSketch, there are no duplicate flow IDs in the hash table, while a flow may be tracked multiple times in EmbedSketch.

Experiment 3: Resource usage in Tofino (Table 2). We compare SeqSketch and EmbedSketch with four state-of-the-art sketch algorithms. We consider four types of resources: stages, ALUs, and VLIW are used for updating sketch values, while PHV carries data across stages. We find that our algorithms consume fewer stages, ALUs, and VLIW than FR, SL, and UM. The reason is that the three algorithms need to update multiple instances (Table 1), while the components in our algorithms require only simple operations. SeqSketch incurs more resource usage than ES because it needs to update additional Bloom Filter and transfer flow records to the controller for evicted entries. EmbedSketch requires fewer resources than others because updating local structures is much simpler (e.g., fewer hash functions for *BF*).

Experiment 4: Bandwidth usage (Figure 13). We measure the ratio of incurred traffic to the traffic in a time interval. The incurred traffic compromises two parts: the evicted flow records and flow IDs during per-packet updates, and the transfer of the sketch at the end of each interval. We see that achieving NZE monitoring incurs less than 0.7% additional bandwidth consumption. Note that existing sketch algorithms only transfer the sketch structures. Although our algorithms additionally transfer flow IDs, the overall bandwidth usage remains limited for two reasons. First, the sketch structures are quite small. Second, we only send evicted flow records that aggregate a considerable number of packets to the controller. When an individual packet is evicted, it will be absorbed by *FS*. Further, *BF* avoids duplicate transfers of flow IDs.

Experiment 5: Recovery time (Figure14). We measure the recovery time for different number of flows. Currently, we use a single thread for recovery. The time is around 60 seconds in the worst case, which is much worse than sketch algorithms. We can optimize it by assigning recovery operations in different CPU cores. Recall that the time interval containing 100 K flows is around 2.5 seconds. Our 36-core server is sufficient



(a) RIP of SeqSketch (b) RIP of EmbedSketch

Figure 15: (Experiment 6) RIP.

to handle all recovery operations. Further, we can speed up with recent distributed machine learning architectures such as TensorFlow. Since solving optimization problem is not our focus, we leave it in the future work.

Experiment 6: RIP (Figure 15). We further examine the RIP of the two algorithms in Figure 15(a) and Figure 15(b), respectively. We observe that RIP remains below 3 in all cases, which is much smaller than that in existing algorithms (§6). The results reveal that SeqSketch and EmbedSketch produce highly orthonormal matrices, which lead to high accuracy when applying compressive sensing reconstruction.

More results. In Appendix, we also present the throughput in software (Experiment 7). We also present the complete accuracy results under different configurations.

7.5 Discussion

Correctness. The correctness of both SeqSketch and EmbedSketch can be derived from compressive sensing. Since we recover per-flow values with standard compressive sensing, the recovered results are close to the true values given the orthonormal matrices produced by the two algorithms (Experiment 3). We leave the formal proof in our future work.

Comparison to existing algorithms. Both the sequential design and embedding design have been used in prior algorithms. For example, ElasticSketch [80] evicts records from a hash table to a sketch; MV-Sketch [77] embeds heavy flows in buckets. Our algorithms are different in four aspects. First, our recovery is based on an optimization framework of compressive sensing. Second, we employ a fractional sketch that increments each counter by a fractional value. Third, we maintain a Bloom Filer to track all flow IDs. Finally, we leverage the controller to reduce the overheads in the data plane. With these design choices, our algorithms achieve near-zero errors.

8 Related Work

Measurement algorithms. Hash tables [1, 2, 52, 59] achieve zero errors but incur excessive resource usage. Some approximate techniques reduce memory usage by addressing only heavy hitters [4, 25, 33, 71]. Sampling techniques [14, 41, 69, 70, 74] selectively discard a portion of traffic to improve resource efficiency. Sketch algorithms [20, 36, 37, 49, 53, 54, 68, 80, 85] employ a compact structure in which multiple flows share a counter. These approximate algorithms usually provide theoretical guarantees to bound the incurred errors. However, the bounds are too loose to apply to all flows, leading to poor accuracy in practice (see §2).

Measurement systems. OpenSketch [82], SCREAM [58] and SketchVisor [35] enhance sketch algorithms in different aspects. Some systems boost performance with TCAM [40, 57, 60]. PacketHistory [32] and Planck [65] mirror traffic to the controller. EverFlow [87] and dShark [81] filter out uninterested traffic with pre-defined rules. mOS [38] and Confluo [42] address monitoring at edges. Studies on query languages [29, 31, 61, 73, 78] empower more fine-grained expressions to tune measurement tasks. TPP [39], MOZART [52], and SwitchPointer [76] combine software and hardware devices to provide both flexibility and programmability for network measurement. Different from these works, our work addresses the algorithmic design for flow monitoring. It is complementary to above system studies.

Compressive sensing for network measurement. CounterBraids [55] demonstrates that sketch and compressive sensing are thematically related, but does not actually apply compressive sensing. [48] applies Least Linear Square method to reconstruct flow values from CountMin Sketch, but does not consider other sketch techniques. [21] shows that sketch algorithms can be formulated as a special kind of compressive sensing. [16, 44, 83] leverage compressive sensing to restore missing values in traffic matrices. [7] uses compressive sensing for tomography. SketchVisor [35] merges its two paths with compressive sensing. In contrast, this paper leverages compressive sensing for NZE monitoring.

9 Conclusion

This paper revisits the theoretical bounds provided by sketch algorithms. We observe that the bounds in existing algorithms are too loose to achieve high accuracy for all flows. We address this problem with compressive sensing. We formulate sketch algorithms as matrices and study their suitability to compressive sensing. The results guide us to design two new algorithms accordingly. The efficiency of the two algorithms demonstrates the power of our methodology. We expect that more algorithms can be designed in the future.

Acknowledgements

We thank our shepherd, Jennifer Rexford, and the anonymous reviewers for their valuable comments. The work was supported in part by National Key R&D Program of China (2019YFB1802600), Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (61802365), and "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (No. LZC0019)".

References

[1] O. Alipourfard, M. Moshref, and M. Yu. Re-evaluating Measurement Algorithms in Software. In *Proc. of HotNets*, 2015.

[2] O. Alipourfard, M. Moshref, Y. Zhou, T. Yang, and M. Yu. A Comparison of Performance and Accuracy of Measurement Algorithms in Software. In *Proc. of ACM SOSR*, 2018.

[3] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. Liu, J. Padhye, B. T. Loo, and G. Outhred. 007: Democratically Finding The Cause of Packet Drops. In *Proc. of USENIX NSDI*, 2018.

[4] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Constant Time Updates in Hierarchical Heavy Hitters. In *Proc. of ACM SIGCOMM*, 2017.

[5] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*, 2010.

[6] P. Bosshart, G. Gibb, H.-s. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. of SIGCOMM*, 2013.

[7] Bowden, Rhys Alistair and Roughan, Matthew and Bean, Nigel. Network Link Tomography and Compressive Sensing. In *Proc. of SIGMETRICS*, 2011.

[8] T. Bu, J. Cao, A. Chen, and P. P. C. Lee. Sequential Hashing: A Flexible Approach for Unveiling Significant Patterns in High Speed Networks. *Computer Networks*, 54(18):3309–3326, 2010.

[9] Caida Anonymized Internet Traces 2018 Dataset. http://www.caida.org/data/passive/passive_dataset.xml.

[10] E. J. Candès et al. The Restricted Isometry Property and Its Implications for Compressed Sensing. *Comptes rendus mathematique*, 346(9-10):589–592, 2008.

[11] E. J. 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.

[12] E. J. Candes and T. Tao. Near-Optimal Signal Recovery From Random Projections: Universal Encoding Strategies? *IEEE Transactions on Information Theory*, 52(12):5406–5425, 2006.

[13] E. J. Candès and M. B. Wakin. An Introduction to Compressive Sampling. *IEEE Signal Processing Magazine*, 25(2):21–30, 2008.

[14] M. Canini, D. Fay, D. J. Miller, A. W. Moore, and R. Bolla. Per Flow Packet Sampling for High-Speed Network Monitoring. In *Proceedings of the First International Conference on Communication Systems and Networks (COMSNETS'09)*, 2009.

[15] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[16] Y.-C. Chen, L. Qiu, Y. Zhang, G. Xue, and Z. Hu. Robust Network Compressive Sensing. In *Proc. of MOBICOM*, 2014.

[17] Chen, Haoxian and Foster, Nate and Silverman, Jake and Whittaker, Michael and Zhang, Brandon and Zhang, Rene. Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis. In *Proc. of SOSR*, 2016.

[18] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. Now Publishers Inc., 2012.

[19] G. Cormode and S. Muthukrishnan. What's New: Finding Significant Differences in Network Data Streams. In *Proc. of IEEE INFOCOM*, 2004.

[20] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[21] G. Cormode and S. Muthukrishnan. Towards an Algorithmic Theory of Compressed Sensing. Technical report, 2005.

[22] G. B. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[23] Data Plane Development Kit. https://dpdk.org.

[24] D. L. Donoho. Compressed Sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306, 2006.

[25] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proc. of ACM SIGCOMM*, 2002.

[26] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High-Speed Links. In *Proc. of ACM SIGCOMM*, 2003.

[27] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[28] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The Analysis of A Near-optimal Cardinality Estimation Algorithm. In *Proc. of AOFA*, pages 127–146, 2007.

[29] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *Proc. of ICFP*, 2011.

[30] Gong, Deli and Tran, Muoi and Shinde, Shweta and Jin, Hao and Sekar, Vyas and Saxena, Prateek and Kang, Min Suk. Practical Verifiable In-network Filtering for DDoS Defense. In *Proc. of IEEE ICDCS*, 2019.

[31] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proc. of ACM SIGCOMM*, 2018.

[32] 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 USENIX NSDI*, 2014.

[33] R. Harrison, Q. Cai, A. Gupta, and J. Rexford. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proc. of ACM SOSR*, 2018.

[34] N. J. Harvey, J. Nelson, and K. Onak. Sketching and Streaming Entropy via Approximation Theory. In *Proc. of FOCS*, 2008.

[35] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proc. of ACM SIGCOMM*, 2017.

[36] 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.

[37] Q. Huang, P. P. C. Lee, and Y. Bao. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Proc. of ACM SIGCOMM*, 2018.

[38] M. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mOS: a reusable networking stack for flow monitoring middleboxes. In *Proc. of NSDI*, 2017.

[39] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *Proc. of ACM SIGCOMM*, 2014.

[40] L. Jose, M. Yu, and J. Rexford. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *USENIX HotICE*, 2011.

[41] S. Kandula and R. Mahajan. Sampling Biases in Network Path Measurements andWhat To Do About It. In *Proc. of ACM IMC*, 2009.

[42] Khandelwal, Anurag and Agarwal, Rachit and Stoica, Ion. Confluo: Distributed Monitoring and Diagnosis Stack for High-Speed Networks. In *Proc. of USENIX NSDI*, 2019.

[43] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic External Memory for Switch Data Planes. In *Proc. of ACM HotNets*, 2018.

[44] L. Kong, M. Xia, X. Liu, M. Wu, and X. Liu. Data Loss and Reconstruction in Sensor Networks. In *Proc. of IEEE INFOCOM*, 2013.

[45] B. Krishnamurthy, S. Sen, Y. Zhang, F. Park, and Y. Chen. Sketch-based Change Detection : Methods , Evaluation , and Applications. In *Proc. of ACM IMC*, 2003.

[46] 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.

[47] LASSO. https://en.wikipedia.org/wiki/Lasso_(statistics).

[48] G. M. Lee, H. Liu, Y. Yoon, and Y. Zhang. Improving Sketch Reconstruction Accuracy Using Linear Least Squares Method. In *Proc. of IMC*, 2005.

[49] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *Proc. of USENIX NSDI*, 2016.

[50] Y. Li, R. Miao, C. Kim, and M. Yu. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *Proc. of ACM CoNEXT*, 2016.

[51] Li, Yuliang and Miao, Rui and Alizadeh, Mohammad and Yu, Minlan. DETER: Deterministic TCP Replay for Performance Diagnosis. In *Proc. of USENIX NSDI*, 2019.

[52] X. Liu, M. Shirazipour, M. Yu, and Y. Zhang. MOZART: Temporal Coordination of Measurement. In *Proc. of ACM SOSR*, 2016.

[53] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *ACM SIGCOMM*, 2019.

[54] 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 ACM SIGCOMM*, 2016.

[55] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter Braids: A Novel Counter Architecture for Per-Flow Measurement. In *Proc. of SIGMETRICS*, 2008.

[56] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[57] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. of ACM SIGCOMM*, 2014.

[58] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *Proc. of ACM CoNEXT*, 2015.

[59] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *Proc. of ACM SIGCOMM*, 2016.

[60] S. Narayana, M. T. Arashloo, J. Rexford, and D. Walker. Compiling Path Queries. In *Proc. of USENIX NSDI*, 2016.

[61] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. of ACM SIGCOMM*, 2017.

[62] OpenvSwitch. http://openvswitch.org.

[63] P4 Language. https://p4.org.

[64] Y. C. Pati, R. Rezaiifar, and P. S. Krishnaprasad. Orthogonal Matching Pursuit: Recursive Function Approximation with Applications to Wavelet Decomposition. pages 40–44, 1993.

[65] 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 ACM SIGCOMM*, 2014.

[66] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network' s (Datacenter) Network. *Proc. of ACM SIGCOMM*, 2015.

[67] B. Schlinker, I. Cunha, Y.-C. Chiu, S. Sundaresan, and E. Katz-Bassett. Internet Performance from Facebook's Edge. In *Proc. of IMC*, 2019.

[68] 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.

[69] 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.

[70] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *Proc. of ACM IMC*, 2010.

[71] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *Proc. of ACM SOSR*, 2017.

[72] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup using Extended Bloom Filter. In *Proc. of ACM SIGCOMM*, 2005.

[73] H. H. Song, L. Qiu, and Y. Zhang. NetQuest: A Flexible Framework for Large-scale Network Measurement. *IEEE/ACM Trans. on Networking*, 17(1):106–119, 2009.

[74] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger. On Unbiased Sampling for Unstructured Peer-to-peer Networks. *IEEE/ACM Trans. on Networking*, 17(2):377–390, 2009.

[75] S. J. Szarek. Condition Numbers of Random Matrices. *Journal of Complexity*, 7(2):131–149, 1991.

[76] P. Tammana, R. Agarwal, and M. Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In *Proc. of USENIX NSDI*, 2018.

[77] L. Tang, Q. Huang, and P. P. C. Lee. MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019.

[78] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever. Stroboscope: Declarative Traffic Mirroring on a Budget. In *Proc. of USENIX NSDI*, 2018.

[79] Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[80] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proc. of ACM SIGCOMM*, 2018.

[81] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *Proc. of USENIX NSDI*, 2019.

[82] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *Proc. of USENIX NSDI*, 2013.

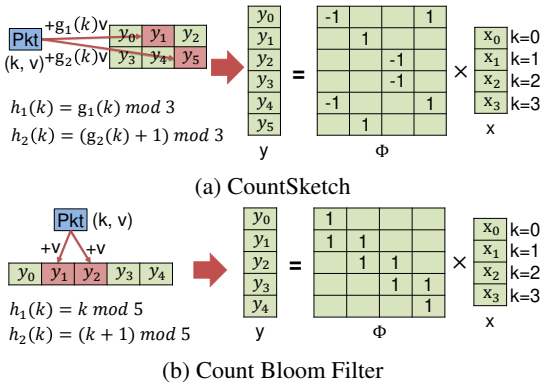
[83] Y. Zhang, M. Roughan, W. Willinger, and L. Qiu. Spatio-Temporal Compressive Sensing and Internet Traffic Matrices. In *Proc. of ACM SIGCOMM*, 2009.

[84] Q. Zhao, A. Kumar, and J. Xu. Joint Data Streaming and Sampling Techniques for Detection of Super Sources and Destinations. In *Proc. of IMC*, 2005.

[85] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, and N. Zhang. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlet. In *Proc. of NSDI*, 2021.

[86] Zheng, Shengbao and Yang, Xiaowei. Dynashield: Reducing the Cost of DDoS Defense Using Cloud Services. In *Proc. of HotCloud*, 2019.

[87] 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 ACM SIGCOMM*, 2015.



(a) CountSketch

(b) Count Bloom Filter

Figure 16: Matrix formulation of CS and CBF.

Appendix A: Sketch-based Sensing

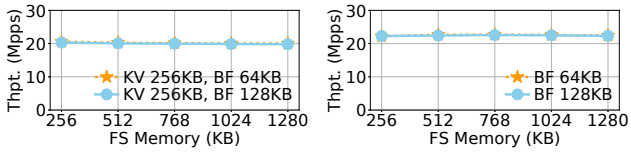
Examples. Figure 16 presents another two examples of the sensing matrices for CS and CBF. As §4.3, we also consider four flows. Figure 16(a) shows a CountSketch (CS). CS has the same structure as CM. However, a packet (k,v) increments its hashed counter in row t by $g_t(k)\cdot v$, where $g_t(k)$ is another function that maps a flow ID to $\{-1,1\}$. Thus, $\Phi_{i,k}$ is either 1 or -1 if flow k hits counter i. Figure 16(b) presents a Bloom Filter. Note that the original Bloom Filter is not a linear mapping, because it maintains an array of bits and performs bitwise OR operations. We extend it to a Counting Bloom Filter (CBF) that replaces the bit array with a counter array. Each counter is updated by v for a packet (k,v) hashed to it. Thus, we set $\Phi_{i,k}=1$ if flow k hashes to counter i.

Appendix B: Implementation

Software version. The software version integrates Open-VSwitch (OVS) [62]. We target two implementations of OVS: one resides in the kernel space, and the other bypasses the kernel via DPDK [23]. In each implementation, we intercept packets in the forwarding module. We put the packet headers in a region of shared memory. A dedicated thread reads the shared memory and updates the sketch (either SeqSketch or EmbedSketch) accordingly.

Hardware version. We implement the hardware version in P4 [63] and target PISA [6] switches. We place the data structures in switch registers, and invoke stateful ALUs to update register values for each packet. However, the limited memory access model of PISA raises two challenges. The first challenge is that each memory access can only manipulate at most 64-bit variables, but in our algorithms, we need to update more than 64 bits of data each time. Second, PISA partitions hardware resources into several stages, each of which is associated with its own ALUs and registers. An ALU can only access the registers belonging to its same stage.

To this end, we tailor SeqSketch and EmbedSketch to fit them into PISA switches. For the first challenge, we sepa-



(a) SeqSketch. (b) EmbedSketch.

Figure 17: Throughput in software.

Table 3: Compressive sensing results with different memory.

Matrix	Recovery	Memory (KB)	(<1e-1)	(<5e-2)	(<1e-2)	(<1e-3)
BM	L1	100	2.01%	1.67%	1.52%	1.52%
BM	L1	200	100%	100%	100%	100%
BM	OMP	100	1.94%	1.62%	1.47%	1.46%
BM	OMP	200	100%	100%	100%	100%
FM	L1	100	26.25%	26.25%	26.25%	26.25%
FM	L1	200	52.50%	52.50%	52.50%	52.50%
FM	L1	300	78.76%	78.76%	78.76%	78.76%
FM	L1	400	100%	100%	100%	100%
FM	OMP	100	14.81%	14.23%	14.06%	14.04%
FM	OMP	200	100%	100%	100%	100%
GM	L1	100	1.94%	1.59%	1.44%	1.44%
GM	L1	200	100%	100%	100%	100%
GM	OMP	100	1.90%	1.60%	1.46%	1.46%
GM	OMP	200	100%	100%	100%	100%
IM	L1	100	1.98%	1.62%	1.48%	1.48%
IM	L1	200	100%	100%	100%	100%
IM	OMP	100	1.98%	1.62%	1.48%	1.48%
IM	OMP	200	100%	100%	100%	100%

rate different types of variables across stages, such that the size of accessed variables in each stage does not exceed the 64-bit limit. For the second challenge, we introduce a few intermediate variables to break the inter-dependencies among variables. More precisely, we store only f and c in the same stage, but replace the variable d with a new variable d'. The new variable d' resides in a stage before f and c. It counts all incoming flows (i.e. $d' = c + d$) and records its value in a metadata field such that it can be shared across stages. The later stage (i.e., that actually maintains f and c) reads d' from the metadata, and determines to perform an eviction operation based on whether $d' - c > c$.

Appendix C: more experiments

Experiment 7: Throughput in software switches (Figure17). We measure the throughput of the two algorithms. We observe that both algorithms keep stable throughput. The throughput of EmbedSketch is higher than that of SeqSketch because its local structures are simpler (see Experiment 3).

Complete results. Table 3 provides the results of classical compressive sensing under different memory settings. Table 4 shows the theoretical configurations of state-of-the-art algorithms with 1% threshold, 1% relative error, and 5% error probability. Table 5, Table 6 Table 7, and Table 8 show the complete results of SeqSketch and EmbedSketch.

Table 4: Theoretical configurations of exiting algorithms.

Algorithm	CU Sketch	Deltoid	ElasticSketch	FlowRadar	NitroSketch
Memory (KB)	312	32500	4438	2115	32672
Algorithm	RevSketch	SeqHash	SketchLearn	SketchVisor	UnivMon
Memory (KB)	58594	32500	32500	2123	32672

Table 5: SeqSketch under different epoch lengths.

Epoch Length (s)	1s	2s	5s	10s	25s
Total Memory (KB)	672	1344	2016	3360	6720
KV Memory (KB)	128	256	384	640	1280
BF Memory (KB)	32	64	96	160	320
FS Memory (KB)	512	1024	1536	2560	5120
(<1e-1)	98.80%	99.51%	98.55%	98.13%	99.95%
(<5e-2)	98.79%	99.50%	98.51%	98.07%	99.95%
(<1e-2)	98.78%	99.50%	98.48%	98.04%	99.95%
(<1e-3)	98.77%	99.49%	98.48%	98.03%	99.95%
Precision (%)	100	100	100	100	100
Recall (%)	99	99	99	99	99
Bandwidth Overhead	0.33%	0.29%	0.20%	0.17%	0.13%

Table 6: EmbedSketch under different epoch lengths.

Epoch Length (s)	1s	2s	5s	10s	25s
Total Memory (KB)	2592	5184	7776	12960	25920
BF Memory (KB)	32	64	96	160	320
FS Memory (KB)	2560	5120	7680	12800	25600
(<1e-1)	98.62%	99.34%	98.46%	98.39%	98.31%
(<5e-2)	98.55%	99.30%	98.38%	98.30%	98.19%
(<1e-2)	98.51%	99.26%	98.32%	98.23%	98.13%
(<1e-3)	98.50%	99.25%	98.31%	98.22%	98.13%
Precision (%)	100	100	100	100	100
Recall (%)	99	99	99	99	99
Bandwidth Overhead	0.81%	0.79%	0.50%	0.41%	0.33%

Table 7: SeqSketch configurations.

Total Memory (KB)	KV Memory (KB)	BF Memory (KB)	FS Memory (KB)	(<1e-1)	(<5e-2)	(<1e-2)	(<1e-3)	Precision (%)	Recall (%)	Bandwidth Overhead
544	256	32	256	37.87%	37.28%	36.81%	36.70%	100	71	0.0854%
800	256	32	512	81.46%	80.80%	80.20%	80.11%	100	92	0.115%
576	256	64	256	43.08%	42.15%	41.39%	41.24%	100	77	0.0854%
832	256	64	512	96.78%	96.66%	96.55%	96.53%	100	99	0.115%
1088	256	64	768	98.28%	98.21%	98.17%	98.16%	100	99	0.144%
1344	256	64	1024	98.89%	98.86%	98.84%	98.84%	100	99	0.173%
1600	256	64	1280	99.04%	99.01%	98.99%	98.99%	100	99	0.202%
640	256	128	256	59.61%	58.19%	57.06%	56.87%	100	87	0.0854%
896	256	128	512	99.63%	99.61%	99.60%	99.60%	100	100	0.115%
1152	256	128	768	99.83%	99.82%	99.82%	99.82%	100	100	0.144%
1408	256	128	1024	99.90%	99.90%	99.90%	99.90%	100	100	0.173%
1664	256	128	1280	99.90%	99.90%	99.90%	99.90%	100	100	0.202%
768	256	256	256	60.12%	58.72%	57.59%	57.41%	100	87	0.0854%
1024	256	256	512	99.97%	99.97%	99.97%	99.97%	100	100	0.115%
1280	256	256	768	99.98%	99.98%	99.98%	99.98%	100	100	0.144%
1536	256	256	1024	100%	100%	100%	100%	100	100	0.173%
1024	256	512	256	60.15%	58.75%	57.62%	57.44%	100	87	0.0854%
1280	256	512	512	100%	100%	100%	100%	100	100	0.115%
1280	256	768	256	60.15%	58.75%	57.62%	57.44%	100	87	0.0854%
1536	256	768	512	100%	100%	100%	100%	100	100	0.115%
1536	256	1024	256	60.15%	58.75%	57.62%	57.44%	100	87	0.0854%
1792	256	1024	512	100%	100%	100%	100%	100	100	0.115%
800	512	32	256	54.65%	53.99%	53.44%	53.34%	100	81	0.109%
1056	512	32	512	93.04%	92.84%	92.68%	92.66%	100	97	0.138%
1312	512	32	768	95.01%	94.87%	94.76%	94.74%	100	98	0.167%
832	512	64	256	67.98%	67.25%	66.69%	66.60%	100	88	0.109%
1088	512	64	512	98.92%	98.89%	98.87%	98.86%	100	100	0.138%
1344	512	64	768	99.38%	99.37%	99.35%	99.35%	100	100	0.167%
896	512	128	256	80.63%	79.98%	79.48%	79.39%	100	94	0.109%
1152	512	128	512	99.88%	99.88%	99.88%	99.88%	100	100	0.138%
1408	512	128	768	99.94%	99.94%	99.94%	99.94%	100	100	0.167%
1024	512	256	256	88.02%	87.58%	87.24%	87.19%	100	96	0.109%
1280	512	256	512	99.98%	99.98%	99.98%	99.98%	100	100	0.138%
1536	512	256	768	99.99%	99.99%	99.99%	99.99%	100	100	0.167%
1792	512	256	1024	100%	100%	100%	100%	100	100	0.196%
1280	512	512	256	100%	100%	100%	100%	100	100	0.109%
1536	512	768	256	100%	100%	100%	100%	100	100	0.109%
1792	512	1024	256	100%	100%	100%	100%	100	100	0.109%
1280	768	256	256	99.91%	99.91%	99.91%	99.91%	100	100	0.105%
1536	768	256	512	100%	100%	100%	100%	100	100	0.134%
1536	768	512	256	100%	100%	100%	100%	100	100	0.105%
1792	768	768	256	100%	100%	100%	100%	100	100	0.105%
2048	768	1024	256	100%	100%	100%	100%	100	100	0.105%
1536	1024	256	256	99.97%	99.97%	99.97%	99.97%	100	100	0.16%
1792	1024	256	512	99.99%	99.99%	99.99%	99.99%	100	100	0.189%
2048	1024	256	768	99.99%	99.99%	99.99%	99.99%	100	100	0.218%
2304	1024	256	1024	100%	100%	100%	100%	100	100	0.247
1792	1024	512	256	100%	100%	100%	100%	100	100	0.16%
2048	1024	768	256	100%	100%	100%	100%	100	100	0.16%
2304	1024	1024	256	100%	100%	100%	100%	100	100	0.16%
2048	1280	256	512	100%	100%	100%	100%	100	100	0.216%
2304	1280	512	512	100%	100%	100%	100%	100	100	0.216%

Table 8: EmbedSketch configurations.

Total Memory (KB)	BF Memory (KB)	FS Memory (KB)	(<1e-1)	(<5e-2)	(<1e-2)	(<1e-3)	Precision (%)	Recall (%)	Bandwidth Overhead
1568	32	1536	72.26%	71.70%	71.21%	71.13%	100	87	0.205%
2080	32	2048	85.62%	85.12%	84.73%	84.67%	100	94	0.259%
2592	32	2560	91.87%	91.63%	91.40%	91.35%	100	97	0.315%
4128	32	4096	97.81%	97.72%	97.64%	97.63%	100	99	0.484%
1600	64	1536	82.09%	81.34%	80.73%	80.62%	100	94	0.21%
2112	64	2048	90.61%	90.37%	90.13%	90.09%	100	96	0.264%
2624	64	2560	95.76%	95.62%	95.51%	95.49%	100	98	0.318%
4160	64	4096	98.14%	98.07%	98.00%	97.99%	100	99	0.485%
1664	128	1536	88.48%	87.80%	87.17%	87.06%	100	98	0.214%
2176	128	2048	94.95%	94.75%	94.53%	94.50%	100	98	0.266%
2688	128	2560	98.51%	98.47%	98.43%	98.41%	100	99	0.32%
4224	128	4096	99.82%	99.81%	99.81%	99.80%	100	00	0.487%
1792	256	1536	89.02%	88.35%	87.70%	87.59%	100	98	0.214%
2048	256	1792	93.83%	93.45%	93.06%	93.00%	100	99	0.24%
2304	256	2048	97.47%	97.31%	97.12%	97.10%	100	100	0.267%
2560	256	2304	99.16%	99.09%	99.03%	99.02%	100	100	0.294%
2816	256	2560	99.75%	99.74%	99.72%	99.72%	100	100	0.321%
3072	256	2816	99.97%	99.97%	99.97%	99.97%	100	100	0.348%
4352	256	4096	100%	100%	100%	100%	100	100	0.487%
2048	512	1536	88.92%	88.24%	87.59%	87.48%	100	98	0.214%
2304	512	1792	93.78%	93.40%	93.02%	92.95%	100	100	0.240%
2560	512	2048	97.54%	97.38%	97.19%	97.17%	100	100	0.267%
2816	512	2304	99.10%	99.04%	98.97%	98.96%	100	100	0.294%
3072	512	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
3328	512	2816	100%	99.99%	99.99%	99.99%	100	100	0.348%
4068	512	4096	100%	100%	100%	100%	100	100	0.488%
2304	768	1536	88.92%	88.24%	87.59%	87.48%	100	98	0.213%
2560	768	1792	93.78%	93.39%	93.01%	92.95%	100	99	0.24%
2816	768	2048	97.54%	97.38%	97.19%	97.17%	100	100	0.267%
3072	768	2304	99.10%	99.03%	98.96%	98.96%	100	100	0.294%
3328	768	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
3584	768	2816	100%	100%	100%	100%	100	100	0.348%
2560	1024	1536	88.91%	88.24%	87.58%	87.47%	100	98	0.214%
2816	1024	1792	93.78%	93.40%	93.02%	92.95%	100	99	0.24%
3072	1024	2048	97.53%	97.37%	97.18%	97.16%	100	100	0.267%
3328	1024	2304	99.10%	99.03%	98.96%	98.96%	100	100	0.294%
3584	1024	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
3840	1024	2816	100%	100%	100%	100%	100	100	0.348%
2816	1280	1536	88.92%	88.24%	87.58%	87.47%	100	98	0.214%
3072	1280	1792	93.78%	93.40%	93.02%	92.95%	100	100	0.24%
3328	1280	2048	97.53%	97.37%	97.18%	97.16%	100	100	0.267%
3584	1280	2304	99.10%	99.03%	98.96%	98.96%	100	100	0.294%
3840	1280	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
4096	1280	2816	100%	100%	100%	100%	100	100	0.348%
3072	1536	1536	88.92%	88.24%	87.58%	87.47%	100	98	0.214%
3328	1536	1792	93.78%	93.40%	93.02%	92.95%	100	100	0.24%
3584	1536	2048	97.53%	97.37%	97.18%	97.16%	100	100	0.267%
3840	1536	2304	99.10%	99.03%	98.96%	98.96%	100	100	0.294%
4096	1536	2560	99.84%	99.83%	99.82%	99.82%	100	100	0.321%
4352	1536	2816	100%	100%	100%	100%	100	100	0.348%