Software Defined Traffic Measurement with OpenSketch

Minlan Yu[†] *Lavanya Jose*[*] *Rui Miao*[†]
[†] *University of Southern California* [*] *Princeton University*

Abstract

Most network management tasks in software-defined networks (SDN) involve two stages: measurement and control. While many efforts have been focused on network control APIs for SDN, little attention goes into measurement. The key challenge of designing a new measurement API is to strike a careful balance between generality (supporting a wide variety of measurement tasks) and efficiency (enabling high link speed and low cost). We propose a software defined traffic measurement architecture OpenSketch, which separates the measurement data plane from the control plane. In the data plane, OpenSketch provides a simple three-stage pipeline (hashing, filtering, and counting), which can be implemented with commodity switch components and support many measurement tasks. In the control plane, OpenSketch provides a measurement library that automatically configures the pipeline and allocates resources for different measurement tasks. Our evaluations of real-world packet traces, our prototype on NetFPGA, and the implementation of *five* measurement tasks on top of OpenSketch, demonstrate that OpenSketch is general, efficient and easily programmable.

1 Introduction

Recent advances in software-defined networking (SDN) have significantly improved network management. Network management involves two important stages: (1) measuring the network in real time (e.g., identifying traffic anomalies or large traffic aggregates) and then (2) adjusting the control of the network accordingly (e.g., routing, access control, and rate limiting). While there have been many efforts on designing the right APIs for network *control* (e.g., OpenFlow [29], ForCES [1], rule-based forwarding [33], etc.), little thought has gone into designing the right APIs for *measurement*. Since control and measurement are two important halves of network management, it is important to design and build a new software-defined measurement architecture. The key challenge is to strike a careful balance between generality (supporting a wide variety of measurement tasks) and efficiency (enabling high link speed and low cost).

Flow-based measurements such as NetFlow [2] and sFlow [42] provide generic support for different measurement tasks, but consume too resources (e.g., CPU, memory, bandwidth) [28, 18, 19]. For example, to identify the big flows whose byte volumes are above a threshold (i.e., heavy hitter detection which is important for traffic engineering in data centers [6]), NetFlow collects flow-level counts for *sampled* packets in the data plane. A high sampling rate would lead to too many counters, while a lower sampling rate may miss flows. While there are many NetFlow improvements for specific measurement tasks (e.g., [48, 19]), a different measurement task may need to focus on small flows (e.g., anomaly detection) and thus requiring another way of changing NetFlow. Instead, *we should provide more customized and dynamic measurement data collection defined by the software written by operators based on the measurement requirements; and provide guarantees on the measurement accuracy.*

As an alternative, many *sketch*-based streaming algorithms have been proposed in the theoretical research community [7, 12, 46, 8, 20, 47], which provide efficient measurement support for individual management tasks. However, these algorithms are not deployed in practice because of their lack of generality: Each of these algorithms answers just one question or produces just one statistic (e.g., the unique number of destinations), so it is too expensive for vendors to build new hardware to support each function. For example, the Space-Saving heavy hitter detection algorithm [8] maintains a hash table of items and counts, and requires customized operations such as keeping a pointer to the item with minimum counts and replacing the minimum-count entry with a new item, if the item does not have an entry. Such algorithms require not only a customized switch chip (or network processor) to implement, but also are hard to change for a better solution in the future. Instead, we should *design a simple, efficient data plane that is easy to implement with commodity switch components, while leaving those customized data analysis to the software part in the controller.*

Inspired by OpenFlow which enables simple and efficient control of switches by separating control and data functions, we design and implement a new *software-defined traffic measurement* architecture *OpenSketch*. OpenSketch provides a generic and efficient measurement solution, by separating the measurement control and data plane functions (Figure 1). Like OpenFlow, OpenSketch keeps the data plane simple to implement and flexible to configure, while enabling flexible and easy programming for different measurement tasks at the controller. OpenSketch's measurement support, together with OpenFlow-like control support, can form a complete measure and control loop for software-defined networking. We expect that OpenSketch will foster more network management solutions using measurement data and theoretical innovations in measurement algorithms.

We made two major contributions in OpenSketch:

First, OpenSketch redesigns the measurement APIs at switches to be both *generic* and *efficient*. Unlike flow-based measurement, OpenSketch allows more customized and thus more efficient data collection with respect to choosing which flow to measure (using both hashing and wildcard rules), which data to measure (more than just byte/packet counters, such as the average flow size), and how to store the measurement data (more compact data structures rather than simple five tuples plus per-flow counters). We design a three-stage data plane pipeline that supports many measurement tasks with simple configurations, operates at high link speed with limited memory, and works with commodity hardware components.

Second, OpenSketch makes measurement programming easier at the controllers by freeing operators from understanding the complex switch implementations and parameter tuning in diverse sketches. We build a measurement library which automatically configures the data plane pipeline for different sketches and allocates the switch memory across tasks to maximize accuracy. The OpenSketch measurement library also makes it easier for operators to apply new theoretical research results of sketches and streaming algorithms upon commodity switch components.

We compare OpenSketch with NetFlow and streaming algorithms using packet traces from CAIDA [41], and show that OpenSketch provides a good accuracy-memory tradeoff. We build an OpenSketch data plane

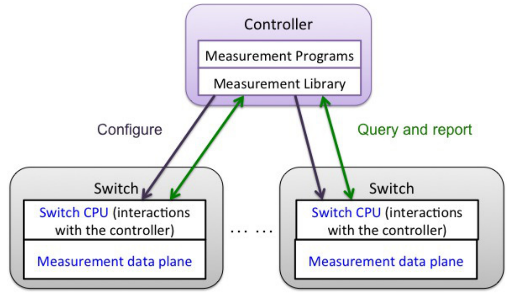


Figure 1: Software defined traffic measurement

prototype on NetFPGA, which shows no additional overhead on switch data plane. Both OpenSketch data and control plane codes are publicly available [5].

Although OpenSketch is sketch-based, it can support a wide variety of measurement tasks because different sketches already support tasks including counting a set of flows, measuring various traffic statistics, and identifying specific flows. In order to make OpenSketch simple enough to implement with commodity hardware and operate at line rate, OpenSketch does not support all the traffic measurement tasks. For example, the OpenSketch data plane does not provide complex data structures (e.g, binary search tree used in some sketches [9]) or directly support all measurement algorithms (e.g., flow size distribution). Instead, we rely on the software in the controller to implement these complex data structures and algorithms using simpler sketches in the data plane.

With our OpenSketch platform, we implement *five* measurement tasks (e.g., heavy hitter detection, flow size distribution calculation) using the measurement library with 15-170 lines of code.

2 Background on Sketches

Sketches are compact data structures used in streaming algorithms to store summary information about the state of packets. Compared to flow-based counters, sketches have two key properties:

(1) Low memory usage: The size of summary information (sketch outputs) is significantly smaller than the input size. For example, the bitmap [21] is a simple sketch that maintains an array of bits to count the number of unique elements (e.g., IP source addresses). We hash each item in a stream of elements to one of the b bits in the bitmap and set the bit to 1. The number of unique elements can be estimated by $b \times \ln(b/z)$ where z is the number of unset bits. Another example is the Count-Min sketch, which maintains a two dimensional array A of counters with width w and depth k. Each entry in the array is initially zero. For each element x, we perform the k pair-wise independent hash functions and increment the counts at $A[i, h_i(x)](i = 1..k)$. To get the frequency for an element x, we just perform the same k hash functions, get k counters from the Count-Min sketch, and report the minimum counter among them.

(2) Provable tradeoffs of memory and accuracy: Sketches often provide a provable tradeoff between memory and accuracy, though the definition of accuracy depends on the actual sketch function. For a bitmap of b bits, the error in the estimated count $\hat{n}$ compared to the real value n is $SD(\hat{n}/n) \approx \sqrt{e^{\rho} - \rho - 1}/(\rho\sqrt{b})$, where ρ is the average number of flows that hash to a bit [21]. In the Count-Min sketch, the relative error in terms of total count is $\varepsilon_{cm} = e \times t \times H_{cm}/C_{cm}$, where H_{cm} is the number of hash functions and the e is Euler's constant, t is the number of bytes per counter, and C_{cm} is its total memory in bytes. Note that the bound is for the worst-case traffic and thus independent of the traffic characteristics. If we have a better understanding of the traffic then we can have a tighter bound for the memory-accuracy tradeoff. For example, for the Count-Min sketch, if we can estimate the distribution skew of the elements in the stream (e.g., a Zipfian parameter of α), we can have a tighter estimation of the error rate $(e \times t \times H_{cm}/C_{cm})^{\max(1,\alpha)}$ [14].

Example measurement solutions built on sketches: Sketches can be used for many measurement tasks such as heavy hitters detection [8], traffic change detection [36], flow size distribution estimation [27], global iceberg detection [25], and fine-grained delay measurement [35]. For example, to count the number of unique sources accessing a web server via port 80, we can simply filter the traffic based on port 80 and then use a bitmap to count the number of unique source addresses. Another example is heavy hitter detection [8], which is important for traffic engineering in data centers [6]. Heavy hitters are those large flows that consume more than a fraction T of the link capacity during a time interval. To identify heavy hitters, we first use a Count-Min sketch to maintain the counts for each flow. Then, we identify potential flows that hashed to heavy counters in a reversible sketch [36], and verify their actual count using the Count-Min sketch.[1]

3 OpenSketch Data Plane

In this section, we first describe the design of OpenSketch data plane. We want to be generic and support various measurement tasks but we also want to be efficient and save switch memory (TCAM, SRAM) and use only a few simple hash functions. Next, we discuss how to implement such a data plane with commodity switch hardware at line rate. Finally, we use several example sketches to show how to configure the simple data plane to meet different requirements.

[1] To insert an element in the reversible sketch, we perform multiple *modular* hash functions on the same key, and increment counters at multiple places. To get the original key, we reverse the modular hash values of the heavy bins, intersect them to get a small set of potential keys. We discuss the technical details of implementing and using reversible sketch in our technical report [44].

3.1 Generic and efficient data plane

A measurement data plane consists of two functions: picking the packets to measure and storing/exporting the measurement data. OpenSketch allows more customized and efficient ways to pick which packets and which data to measure by leveraging a combination of hashing and wildcard rules. OpenSketch also allows more flexible collection of measurement data by breaking the tight bindings between flows and counters. It reduces the amount of data to store and export using more compact data structures.

Picking the packets to measure: OpenSketch shows that a combination of hashing and classification can support a wide variety of ways of picking which packets to measure. Hashes can be used to provide a compact summary of the set of flows to measure. For example, to count the traffic to a set of servers, we can use hashing to provide a compact representation of the set of servers (e.g., Bloom filter). To count the number of redundant packets with the same content, we can hash on the packet body into a short fingerprint rather than store and compare the entire packet body every time. Hashes also enable a provable accuracy and memory tradeoff.

Classification is also useful for focusing on some specific flows. For example, a cloud operator may need to identify the popular flows from a specific tenant or identify the DDoS attack targets within a list of web servers. Therefore, we need a classification stage to measure different flows with different number of counters or with different levels of accuracy. For example, if there is too much traffic from the IP address 192.168.1.1, we can filter out packets from 192.168.1.1/32 i.e. use one counter to count traffic from the specific IP and another to count remaining traffic of interest from the subnet 192.168.1.0/24. For classifying flows, we can specify wildcard rules that match packets on flow fields and allow some bits in the flow fields to be "don't care". For example, the rule can match packets on source IP prefix 192.168.1.0/24, where the lower 8 bits are "don't care" bits.

Storing and exporting the data: OpenSketch uses a small table with complex indexing. Each entry in the table only contains the counters without the flow fields. These counters can be associated with different entities like a microflow, a wildcard flow, or even a hash value. In this way, OpenSketch allows more flexible data collection with much less memory than traditional flow-based

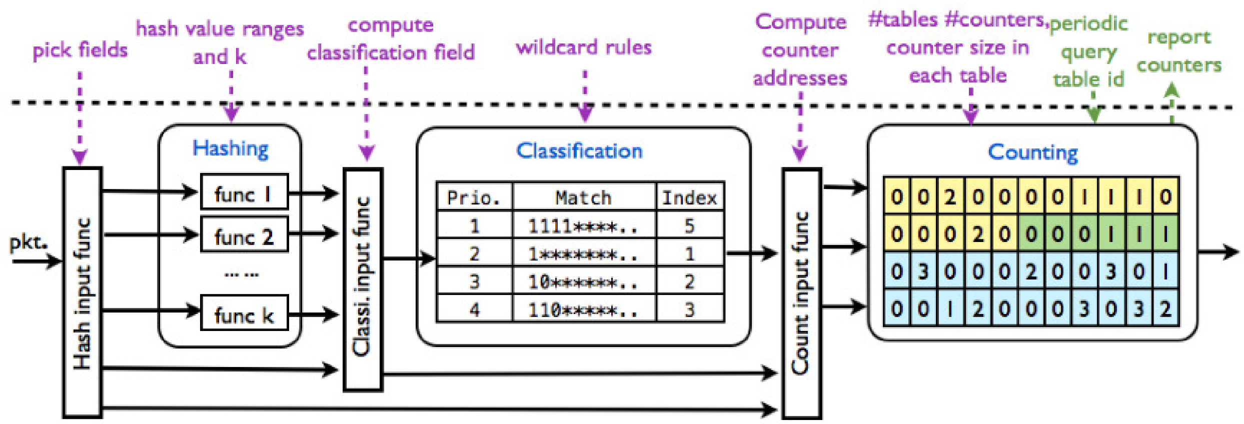


Figure 2: OpenSketch switch data plane

tables. Moreover, OpenSketch can easily export the table to the controller with small bandwidth overhead.

To get such flexibility and memory saving, OpenSketch requires more complex indexing using the hashing and classification modules. Fortunately, these complex indexes are easy to calculate using commodity switch components. For those measurement tasks that need to identify specific flows, the controller can maintain the mappings between counters and flows for classification-based indexing, or reverse engineer the flows from the hash values for hash-based indexing (e.g., using reversible hashing [36]).

OpenSketch data plane: OpenSketch data plane has three stages: a hashing stage to reduce the measurement data, a classification stage to select flows, and a counting stage to accumulate traffic statistics (Figure 2). We use the bitmap as an example to show how the data plane works. Suppose we use the bitmap to count the number of unique source IP addresses to a given destination subnet (say 192.168.1.0/24): First, the hashing stage picks the packet source field and calculates a single hash function. Next, the classification stage picks the packet destination field and filters all the packets matching the rule ($dst: 192.168.1.0/24 \rightarrow 1$). Each rule has an index field, which can be used to calculate the counter location in the counting stage. For example, those packets in the subnet get the index "1", which means counting; the other packets get the default index "-1" and are not counted. Finally, the counting input function calculates the index of the bit to be updated using the hash value of the source field. The corresponding position in the counting table for the bitmap is marked as 1.

3.2 Build on existing switch components

OpenSketch data plane can be easily implemented with commodity switch components:

A few simple hash functions: OpenSketch relies on hashing to pick the right packets to measure with provable memory and accuracy tradeoffs. However, sketches may need a different number of hash functions or different types of hash functions, and may operate on different packet fields. For example, the Count-Min sketch requires k (e.g., 3) pairwise independent hash functions. On the other hand bitmaps [43] and the PCSA sketch [22] require a truly random hash function (i.e. each item is hashed uniformly on its range and completely independently of others).

Fortunately, our experiences of implementing various sketches show that 4-8 three-wise or five-wise independent hash functions are enough for many measurement requirements, and can be implemented efficiently in hardware [34, 39]. Moreover, simple hash functions can make use of the entropy in the traffic to approximate even truly random hash functions well [31].

We can also reduce the number of hash functions by allowing multiple sketches to share the same set of hash functions.

A few TCAM entries for classification: Classification can be easily implemented with high-speed memory TCAMs (Ternary Content-Addressable Memory), which already exist in today's switches to store ACLs (Access Control Lists). TCAMs can match packets with multiple rules *in parallel* and perform actions based on the rule with the highest priority. Each rule contains the matching fields (including 0's, 1's, and"don't care" bits) and actions (such as incrementing a counter and pointing to a SRAM address).

TCAMs are expensive and power hungry, and thus only have a limited number of entries in most switches (e.g., at most thousands of entries [16]). Since OpenSketch leverages a combination of hashing and classification to select packets, it does not maintain individual flow entries in the TCAM and thus significantly reduces the number of TCAM entries to support most measurement tasks. In addition to simple wildcard matching on packet fields, we allow TCAM entries to match on hash values and leverage the "don't care" bits to perform other operations such as set checking. We will show our detailed design for supporting different sketches in the next subsection.

Flexible counters in SRAM: We store all the counters in the SRAM, because SRAMs are much cheaper, more energy-efficient, and thus larger than TCAMs. Leveraging the provable memory-accuracy tradeoffs in sketches, we can make sure that the sketches always fit in the switch SRAM independent of the traffic pattern.

However, different sketches require different numbers and sizes of counters: the bitmap contains an array of 0's and 1's, while the Count-Min sketch [13] contains several counters for a single packet. We introduce a list of *logical* tables in the SRAM (e.g., in Figure 2 we show three logical tables of different colors). These tables can represent different counters in the same sketch (e.g., k tables for the Count-Min Sketch) or different sketches in the same SRAM. Since all the counters stored in the SRAM can be easily accessed by their addresses, we use a single physical address space to identify the counters in all the logical tables. Based on the table *id* and the relative counter position in the table, we can easily locate the counter position in the SRAM. On those SRAMs that have 6-8 read/write ports [17], these counters can even be updated at the same time.

3.3 Supporting diverse sketches

OpenSketch data plane can support a wide variety of sketches by leveraging a smart combination of hashing, classification, and counting. Here we only discuss a few sketches, for which an implementation in OpenSketch is not obvious. We show more sketch examples such as hash tables, Count-Min sketches, and wildcard filters in [44]. The goal of this subsection is not to show the detailed tricks of implementing a specific sketch, but to show the power of the simple OpenSketch data plane in implementing many complex sketches.

Bit checking operations: Many sketches require more complex bit checking operations than simply comparing the packet fields to a pre-defined rule. For example, the Bloom filter, which is used to filter packets based on a predefined set, requires checking if k positions are 1 or not (based on k hash values calculated from the packets). The DFAs (Deterministic Finite Automaton) [10, 38] and regular expressions [30], which are often used for network intrusion detection, require converting one packet state to another.

To implement such sketches in OpenSketch data plane, we can leverage the hashes and the wildcard bits in TCAMs. For example, to check if a packet's source port belongs to a predefined set of source ports with Bloom filters, we first calculate the Bloom filter array B of 0's and 1's (e.g., 0001101101) of the pre-defined set. Next, we calculate the k hash functions on each incoming packet's source port and generate the packet's array P (e.g., 0001000001). Now we need to check if all the 1's positions in P are also 1's in B. Although such complex bit checking is not supported by TCAM, we can match P with B^*, where B^* replaces all the 1's in B with $*$ (e.g., B^* =000**0**0*) [23]. Then we can match P against B^*. The 0 in B^* correspond to bits that were not set by any packet in B. If P has a 1 where B^* has a 0, then we can conclude that P is not in B. But if P matches the TCAM entry B^*, there is a chance that P is in B, and we say the packet matches the Bloom filter.

Picking packets with a given probability: Many sketches require picking packets with different probabilities. For example, the PCSA sketch (Probabilistic Counting with Stochastic Averaging) [22] provides a way to count the number of unique values of a header field(s) The basic idea is to sample packets into different bins with power-of-two ratios (e.g., 1/2, 1/4, ...$1/2^n$). If there's a packet that falls in the bin i, then that means there are at least 2^i different values. Other streaming algorithms may require sampling packets at a given rate (not necessarily power-of-two) to reduce the measurement overhead.

To support this probabilistic packet selection using only simple uniform hash functions in OpenSketch, we choose to combine these hashes with a few TCAM entries. For example, to implement power of two probabilities, we first calculate the hash value for a packet field using the uniform hash function. Next, we compare the hash value with TCAM rules such as $R_1 : 0*** ... \rightarrow 1$; $R_2 : 10**... \rightarrow 2$; $R_3 : 110*... \rightarrow 3$; etc. There is a 1/2 chance that a hash value matches rule R_1, 1/4 for R_2, an so on. We can further combine these rules to implement other more complex probabilities.

Picking packets with different granularity: Many streaming algorithms (such as flow size distribution [27], counting distinct destinations with the multi-resolution bitmap [21], and latency tracking with arbitrary loss [26]) often make tradeoffs between flow space coverage and accuracy. If they cover a smaller flow space, they can devote more counters to more accurately measure the covered space, but the measurement result is less representative for the entire flow space. If they cover a larger flow space, the result is more representative but less accurate. Therefore, many algorithms require different levels of coverage in the flow space. For example, the algorithm to measure flow-size distribution [27] hashes on the packets and maps the hash value to different sizes of ranges ([0,1/2],[1/2, 3/4]), ...) and measures the flow sizes within these ranges. The multi-resolution bitmap [21] uses a similar idea to count the number of unique elements in each range. However, all these algorithms assume customized hardware.

Instead, we design a new multi-resolution classifier that can be easily implemented with TCAM and used in different measurement programs. The multi-resolution classifier uses $\lg(n)$ TCAM entries to represent different ranges in the flow space, where n is the range of the flowspace: $R_1 : 0***... \to 1$; $R_2 : 10**... \to 2$; $R_3 : 110*... \to 3$; etc. Operators simply need to glue the multi-resolution classifier before their other measurement modules to increase the accuracy of their measurements with a good flow space coverage.

4 OpenSketch Controller

With OpenSketch providing a simple and efficient data plane, operators can easily program measurement algorithms and even invent new measurement algorithms by gluing different sketches together. The OpenSketch controller provides a sketch library with two key components: A sketch manager that automatically configures the sketches with the best memory-accuracy tradeoff; and a resource allocator that divides switch memory resources across measurement tasks.

4.1 Programming measurement tasks

Although OpenSketch is sketch-based, it can support a wide variety of measurement tasks because: (1) There are already many sketches for tasks ranging from counting a set of flows, measuring various traffic statistics, to identifying specific flows. (2) Even when some traffic characteristics are not directly supported by sketches, we can still install simpler sketches and implement the complex data analysis part in software in the controller.

As shown in Table 1, we have implemented several measurement programs by simply gluing building blocks together. The algorithms to measure flow size distribution and to count traffic leverage different sketches (e.g., multi-classifier sketches and Bloom filters) to pick the right packets to measure. The heavy hitter detection and superspreader/DDoS detection algorithms leverage Count-Min sketches and k-ary sketches to count specific traffic characteristics. Although there are no sketches that directly measure traffic changes or flow size distribution, we can still rely on sketches to get basic counters (e.g., flow size counters) and leave it to the controller to analyze these counters (e.g., calculate the distribution).

We take the superspreader/DDoS detection as an example to show how to use OpenSketch to implement a measurement task. Superspreaders are those hosts that send packets to more than k unique destinations during a time interval. The goal of superspreader detection is to detect the sources that send traffic to more than k distinct destinations, and ensure that the algorithm does not report the sources with $\leq k/b$ destinations, with high probability $\geq 1-\delta$ [40]. The streaming algorithm for detecting superspreaders [40] samples source-destination pairs and inserts them into a hash table, which requires a customized ASIC to implement in the data plane.

Combining Count-Min sketch and bitmap. Given the building blocks in the OpenSketch measurement library, we implement this superspreader algorithm using a combination of the bitmap, Count-Min sketch, and reversible sketch. Ideally, we would like to use a sketch to count the number of *unique* destinations for each source. However, the Count-Min sketch can only count the number of packets (not unique destinations) for each source, while the bitmap can only provide a single count of the total number of unique destinations but not individual counters for each source. Therefore, we combine the Count-Min sketch and the bitmap to count the number of unique destinations that each source sends: We replace the normal counters in Count-Min sketch with bitmaps to count the unique number of destinations instead of the number of packets. Since the superspreaders algorithm requires us to report a source if the number of retained destinations is more than r, we need a bitmap that can count up to around r destinations. Similarly, we combine reversible sketches with bitmaps to track the superspreader sources.

Sampling source-destination pairs to reduce memory usage: One problem of simply using the sketches is that the trace can still have a large number destinations, leading to large memory requirement to get a reasonable accuracy. To reduce memory usage, we choose to sample and measure only a few source-destination pairs. Similar to the streaming algorithm [40], we sample the packet at rate c/k, and report all source IPs that have more than r destinations, where c and r are defined as a constant, given b and δ as shown in Figure 2 in [40]. For example, when we set $b = 2$ and $\delta = 0.2$, we get $r = 33$ and $c = 44.83$.

Although we use the same sampling solution as the work in [40], there are three key differences: (1) We use Count-Min sketches instead of hash tables, which reduces memory usage with a slightly higher error rate. Count-Min sketches are especially beneficial when there are only a few heavy sources (contacting many destinations) and many light sources (contacting a few destinations), because there are fewer collisions between the heavy and light sources. (2) the work in [40] uses 32 bits to store destination IPs for each pair of source and destination while the bitmap stores only $O(r)$ bits totally. If we set a threshold $r = 33$, we only need a bitmap of

Measurement Tasks	Definitions	Building Blocks
Heavy Hitters [13]	Identify large flows that consume more than a fraction T of the link capacity during a time interval	*Count-Min sketch* to count volume of flows, *reversible sketch* to identify flows in Count-Min with heavy counts
Superspreader/DDoS	A k-superspreader is a host that contacts more than k unique destinations during a time interval. A DDoS victim is a host that is contacted by more than k unique sources.	*Count-Min sketch* to estimate counts for different sources, *bitmap* to count distinct destinations for each *Count-Min sketch* counter, *reversible sketch* to identify sources in *Count-Min sketch* with heavy distinct counts
Traffic changes detection [36]	Identify flows whose absolute contribution to traffic changes are the most over two consecutive time intervals	*k-ary sketch* and *reversible sketches* for consecutive intervals to identify heavy changes
Flow size dist. [27]	Indicate the portion of flows with a particular flow size	*multi-resolution classifier* to index into *hash table* at right granularity; *hash table* to count volume of flows mapped to it
Count traffic	Count the distinct number of source addresses who send traffic to a set of destinations	*Bloom filter* to maintain the set of destinations; *PCSA* to maintain distinct count of sources

Table 1: Implementing measurement tasks using the OpenSketch library

```
for (index in rev_sketch.bins) {
    count = distinct_count.get_count(index, rev_sketch)}
flow_array = rev_sketch.get_heavy_keys(counts, r)
for (flow in flow_array) {
    index_list = count_min.get_index(flow)
    count_list = distinct_count.get_count(index_list, count_min)
    if (all count in count_list > r) output key}
```

Figure 3: The querying function in superspreader detection

149 bits to store all pairs of sources and destinations for each Count-Min counter. (3) It is hard to implement the original streaming algorithm [40] with commodity switch components. This is because the algorithm requires customized actions on the hash table (e.g., looking up the src-dst pair in the hash table, inserting a new entry if it cannot find one and discarding the packet otherwise, removing entries if the hash table is full, etc.).

Querying in the control plane. Figure 3 shows the data analysis part of the superspreader detection program. We first look at the reversible sketch to identify the sources with counts more than r using the distinct counts from the bitmaps and compile a list of heavy bins. The reversible sketch "reverse-hashes" these bins to identify potential superspreader sources. The reversible sketch uses modular hash functions and doesn't provide any accuracy guarantees for the sources. So we next query the Count-Min sketch to verify the counts of the candidate sources. If the count is above the threshold r, then we report the source as a superspreader.

Although we have shown several examples that operators can easily program in OpenSketch, there are still many open questions on how to provide full language support for many other measurement programs. We leave this for future work.

4.2 Automatic config. with sketch manager

Picking the right configurations in the measurement data plane is notoriously difficult, because it depends on the available resources at switches, the accuracy requirements of the measurement tasks, and the traffic distribution. To address these challenges, OpenSketch builds a sketch manager: The sketch manager automatically picks the right sketch to use given the measurement requirements and configures the sketches for the best accuracy given the provable memory-accuracy tradeoffs of individual sketches and the relations across sketches. Furthermore, the sketch manager may automatically install new sketches in the data plane to learn traffic statistics to better configure the sketches.

We take the superspreader problem as an example:

Picking the right sketch for a function. Given the provable tradeoffs between error rate and memory size, the sketch manager automatically picks the right sketch if there are multiple sketches providing the same functions. In superspreader detection, given memory size m bits and the threshold r, there are two common data structures for distinct counters: the PCSA whose error rate is $0.78\sqrt{\lceil \log r \rceil / m}$, and the bitmap whose error rate is $\sqrt{(e^{r/m} - r/m - 1)/(r^2/m)}$. The operator can simply use "distinct counter" as the virtual sketch in his program. The sketch manager can automatically pick the right sketch of distinct counters to materialize the virtual sketch. For example, if there are $m = 149$ bits for distinct counters and $r = 33$, the error rate is 14.5% for PCSA, and 6% for bitmap. Therefore, the manager picks bitmap for the superspreader problem.

Allocating resources across sketches. In OpenSketch, operators can simply configure the sketches required for their programs without detailed configurations. The sketch manager can automatically allocate resources across sketches within the measurement program, given the memory-error tradeoffs of individual sketches. For the superspreader example, the manager automatically formulates the following optimization problem: configuring the sketch parameters (the number of bits in each bitmap m, the number of hash functions H, and the number of counters for each hash function M in the Count-Min sketch), to minimize the error rate of the superspreader detection, given the total memory size C_{ss}, the threshold r, the sampling ratio c/k, and the estimated number of distinct src-dst pairs N.

$$
\begin{aligned}
Min \quad & \varepsilon_{ss} = (\varepsilon_{bm} r + \varepsilon_{cm} \times N \times c/k)/r && (1)\\
s.t. \quad & \varepsilon_{bm} = \sqrt{(e^{r/m} - r/m - 1)/(r^2/m)} && (2)\\
& \varepsilon_{cm} = e(1 + \varepsilon_{bm})/M_{cm} && (3)\\
& C_{ss} = H_{cm} M_{cm}\, m \le C_{total} && (4)\\
& m_{fillup} \le m \le m_{max} && (5)
\end{aligned}
$$

Eq.(1) describes the goal of the optimization problem, which is to minimize the error rate of the combination of Count-Min Sketch and bitmap, which is at most $(\varepsilon_{bm} r + \varepsilon_{cm} \times N \times c/k)/r$ for counts $< r$ as proved in [24]. Eq.(2) and Eq.(3) define the error rate of the bitmap and the Count-Min sketch. Eq.(4) describes the memory constraints: The combination of Count-Min sketch and bitmap should take less memory than that available at the switch. Eq.(5) bounds the number of bits m used in bitmap. As suggested in [43], we set m_{fillup} to be the minimum m such that $m \ge 5\sqrt{e^{r/m} - r/m - 1}$, which is the minimum size to guarantee the bitmap doesn't fill up with high probability. We also set $m_{max} = 10r$, which is large enough so that the relative error is less than 1% for distinct counts up to $r = 100$.

Note that we do not consider the reversible sketches in the optimization. This is because the reversible sketch usually has a fixed size of 5 hash tables each with 4096 (distinct) counters as suggested by [36]. In our evaluations with real packet traces we found that for $\delta = 0.2$, 2 reversible sketches of 98-bit bitmaps (a total of 0.5 MB) was often large enough to not miss any sampled superspreaders (thus the false negative rate is caused by sampling, same as the streaming algorithm). In addition, the false positives of reversible sketches are not important because we always check with Count-Min sketch which provides more accurate counts.

We can easily solve the optimization problem to configure sketches automatically. Suppose we want to find $k = 200$ superspreaders with $b = 2$ and a probability of at most $\delta = 0.2$ false positives and negatives. We set the sampling rate at $c/k = 44.83/200$ and report sources with at least $r = 33$ distinct destinations retained. Suppose the the maximum number of distinct source destination pairs on the link is $N = 400,000$, and we have 1900KB available for the combination of Count-Min sketch and bitmap. Then solving the optimization problem, we find that we can minimize the error when the bitmap has $m = 46$ bits, the CM has 3 hash functions and 109,226 counters per hash function.

Installing new sketches to learn traffic statistics. As shown above, some memory-accuracy tradeoffs require understandings of traffic characteristics to have a more accurate configurations of the sketches. For example, in the superspreader example, we need to have an estimation of the maximum number of source-destination pairs N. In this case, OpenSketch requires operators to given a rough estimation based on experiences (e.g., operators can simply give n^2 for a network of n nodes.)

The sketch manager can then automatically install new sketches to help understand such traffic characteristics. For example, to count the unique pairs of source and destination, the manager installs a new distinct counter and periodically checks its value. If the difference between the counting result and the original estimation is above a pre-defined threshold, then the manager re-optimizes the resource allocation problem.

Though we described the sketch manager in the context of finding superspreaders, functions like picking the right sketch and allocating resources across sketches (which don't interact in complex ways) can easily be automated for other measurement tasks too.

4.3 Resource allocation across tasks

When OpenSketch supports multiple measurement tasks simultaneously, it is important to optimize the accuracy of the measurement tasks given the limited resources in the data plane (e.g., the number of hashing modules, the TCAM and SRAM sizes). Therefore, we build a resource allocator in OpenSketch which automatically allocates resources across measurement tasks. Operators simply need to specify the relative importance of accuracy among the tasks. For example, to allocate resources across heavy hitter and superspreader detection, operator can simply specify a weight $\beta \in [0,1]$. $\beta = 1$ means operators care most about heavy hitter detection and $\beta = 0$ means the other way.

The challenge is to allocate resources across tasks without considering the detailed implementation of each task. We propose to modularize the resource allocation problem using optimization decomposition. The basic idea is to introduce a price λ to indicate the relative resource usage and then leave it to each task to optimize their own accuracy based on the resource allocation.

For example, operators may want to detect heavy hitters using a Count-Min sketch and to detect superspreaders using a combination of Count-Min sketch and bitmaps simultaneously. We first formulate the main allocation problem: Given the total switch SRAM memory C_{total}, the resource allocator allocates the memory across the two measurement tasks (Eq. (7)) to minimize the weighted error (Eq. (6)). For Superspreaders, the error rate ε_{ss} and space used by the building blocks C_{ss} are as described in 1-4. For heavy hitters the error rate is $\varepsilon_{hh} = e/M_{cm}^{hh}$ and the space used is $C_{hh} = H_{cm}^{hh} \times M_{cm}^{hh}$, where H_{cm}^{hh} is the number of hash functions and M_{cm}^{hh} is the number of counters per hash.

$$\begin{aligned} Min \quad & \varepsilon_{all} = (1-\beta)\varepsilon_{ss} + \beta\varepsilon_{hh} && (6)\\ s.t. \quad & C_{ss} + C_{hh} \le C_{total} && (7)\\ given \quad & \beta, C_{total} \end{aligned}$$

Next, we solve the resource allocation problem in a modularized way by first converting the problem into its dual problem:

$$\begin{aligned} Max \quad & \min((1-\beta)\varepsilon_{ss} + \lambda(C_{ss} - C_{max}/2)) \\ & + \min(\beta\varepsilon_{hh} + \lambda(C_{hh} - C_{max}/2)) && (8)\\ s.t. \quad & \lambda \ge 0 \end{aligned}$$

We can then easily decompose the dual problem into two sub-problems corresponding to the two tasks superspreader and heavy hitter detection (The two sub problems are omitted for brevity). Iteratively, the master problem sets a price λ for the space used the two tasks. Each task minimizes its objective at the given price, and the master problem updates its price according to how much extra space was used by the tasks (using the subgradient method). When the price eventually converges to the optimal value, we get the resource allocation across the two tasks and the parameter configurations for each task.[2]

In general, the optimization decomposition technique allows the resource allocator to allocate space efficiently between two tasks without any knowledge of how they're implemented, it only needs to know their relative importance to the operator.

5 Prototype Implementation

Figure 4 shows the key components in our prototype: In the data plane, we implemented the three-stage pipeline (hashing, classification, and counting) on NetFPGA; in the control plane, we implemented the OpenSketch library which includes a list of sketches, the sketch manager, and the resource allocator.

[2]Note that such optimization decomposition only works for convex functions. But the superspreader problem is not convex in terms of number of counters M_{cm}^{ss} *and* the bitmap size m. We work around this by iterate the size of bitmap from m_{fillup} to m_{max}, and thus make the problem convex with a fixed m.

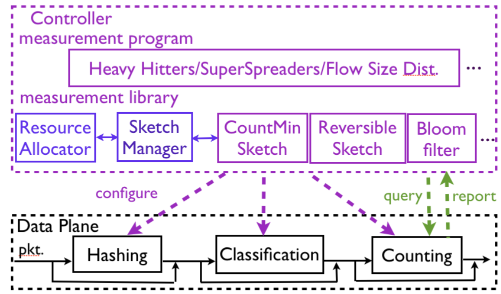
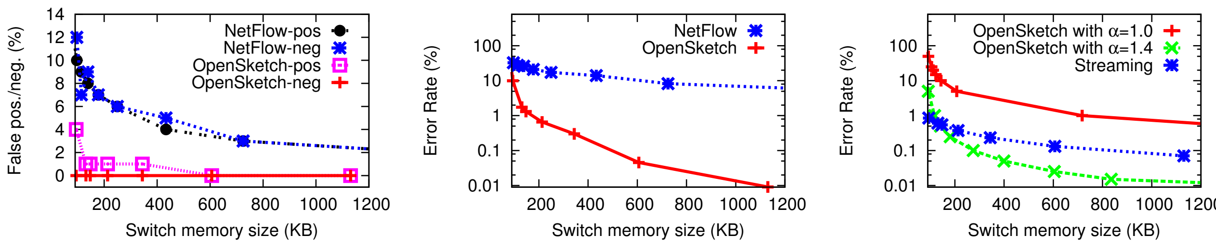


Figure 4: OpenSketch Architecture

Data plane: We implement a OpenSketch switch prototype on NetFPGA to understand its efficiency and complexity in real hardware deployment. We insert our measurement modules into the reference switch pipeline, including Header Parser, Hashing, Wildcard Lookup, and SRAM Counter. Since we simply pull packet headers to collect the statistics without changing the packet bus, there is no effect on packet forwarding latency and throughput. As a packet enters, the Header Parser pulls the related fields from the packet header, which are then hashed by hash functions in parallel. We then pass the packet header and hash values to the Wildcard Lookup, where we implement wildcard rule matching in parallel. For each matched rule, we update the corresponding counters in the SRAM Counter. We use the entire 4.5 MB on-board SRAM to store the counters. To answer the OpenSketch queries from the controller, we implement a userspace software to query these counters via IOCTL calls, which then collects all the counters through PCI interface.

Controller: We implement *seven* sketches in c++ in the controller, including bitmap, PCSA sketch, hash table, multi-resolution classifier, bloom filter, count-min sketch, and reversible sketch. Each sketch has two functions for the measurement programs to use: *configure* to specify the packet fields, the memory constraint, and the number of hash functions to use and *query* to periodically get the statistics. We also implement the sketch manager to configure these sketches and the resource allocator to divide resources across measurement tasks.

The measurement program in the controller can periodically query the sketches about the statistics (e.g., the distinct counts from bitmap, the flows from reversible sketches). According to the received statistics, the measurement programs may install new sketches or change the accuracy requirements accordingly. The sketches automatically queries the data plane to get the counters in order to generate the right statistics to the measurement program. Each time after reporting the counters, the OpenSketch data plane resets the counters as zero in order to monitor the traffic statistics at the next measurement interval. The history statistics are maintained in the controller and written to the disk if necessary.



(a) False positives/negatives of OpenSketch and NetFlow

(b) Error rates of OpenSketch and NetFlow

(c) Theoretical error rates of OpenSketch and a space-saving streaming algorithm

Figure 5: Heavy hitter detection with OpenSketch, NetFlow, and space-saving streaming algorithm

6 Evaluation

To evaluate the balance between generality and efficiency, we use packet-trace driven simulations to compare OpenSketch with NetFlow and other streaming algorithms for several measurement tasks using CAIDA packet traces. To evaluate the feasibility of implementing OpenSketch with commodity switch components, we run stress tests with our OpenSketch prototype. Both our simulator and prototype are available at [5].

6.1 Data plane generality and efficiency

Evaluation setup We build a trace-driven simulator of an OpenSketch implementation of heavy hitters and superspreaders, and compare it to NetFlow with packet sampling, and a superspreader streaming algorithm [40]. We use a one-hour packet trace collected at a backbone link of a Tier-1 ISP in San Jose, CA, at 12pm on Sept. 17, 2009 [41]. We collect the counters at the end of every measurement interval and reset the counters to measure the next interval. We configure the parameters in our evaluation based on recommended values from literature [20, 37]. We set the measurement interval as 5 seconds and run for 120 measurement intervals to ensure the metrics converge.

OpenSketch provides a better memory-accuracy tradeoff than NetFlow. Since both OpenSketch and NetFlow provide general support for diverse measurement tasks, a natural question arises: which one works better? We compare the two using the heavy hitter detection problem because NetFlow, which uses packet sampling, can easily catch large flows and is expected to work well.

We set the threshold T as 0.5% of the link capacity to find a few large senders in the traffic with a heavy-tail distribution. The sketch-based heavy hitter algorithm uses two building blocks: the Count-Min sketch and the reversible sketch, which are described in [44]. These sketches are automatically configured by the sketch manager. We conservatively assume each counter (in both sketches) uses 4 Bytes, though in practice for counting up to $N = 2$ million packets, a counter needs only $lg(N) < 3$ bytes. For NetFlow with packet sampling, the operator can set the sampling rate according to how much memory is available at the router. We conservatively assume that NetFlow uses 32 Bytes per flow entry (as in [20]) and count the actual number of flow entries used by NetFlow.

Figure 5 (a) shows the false positives and false negatives for identifying heavy hitters. OpenSketch has no false-negatives with 85KB memory, and no false positives when the switch has 600KB memory. In contrast, NetFlow needs 724KB memory to achieve 3% false positives and 3% false negatives. Figure 5 (b) compares the error rate in estimating the volume of detected heavy hitters. Like in [20], we define the error rate as the relative error (the average error of each heavy hitter's volume) divided by the size of the threshold for fair comparison. We can see that with 600KB memory, the error rate for OpenSketch is 0.04%, which is far less than NetFlow.

OpenSketch achieves comparable accuracy to the streaming algorithms, while providing generality. Although OpenSketch supports most sketch-based streaming algorithms, it does not support those streaming algorithms that require complex actions in the data plane. We compare our OpenSketch-based algorithm with these streaming algorithms to understand the tradeoff between accuracy and generality.

Heavy hitter detection: We first compare the two theoretical bounds of OpenSketch-based heavy hitter algorithm and the space-saving streaming algorithm proposed in [8].[3] Figure 5 (c) shows that to achieve the same 0.5% error rate, OpenSketch requires 1354KB memory while the streaming algorithm only takes 160KB. However, OpenSketch can further improve its accuracy when it understands the traffic skew (i.e., the Zipfian parameter α in Sec 2) from either operators or by installing a hash table (similar to the flow size distribution problem in Table 1). For example, if OpenSketch knows that the skew

[3] This is the error with 95% confidence.

parameter is $\alpha = 1.4$, it only needs to configure 142KB memory, which is less than the streaming algorithm.

Superspreader detection: We also compare the accuracy of OpenSketch with the one-level superspreader detection algorithm [40] in Figure 6.[4] We set $k = 200$, $b = 2$, and $\delta = 0.05$. We conservatively assume it takes 8 Bytes to store each src-dst pair in a hash table and 6 Bytes to store each source-counts pair in a second hash table The total space used by the two hash tables for the sampled source destination pairs is the memory size (assuming no hash conflicts).

The streaming algorithm requires an average of 1MB memory to achieve a false positive rate of 0.1% and a false negative rate of 0.3%. For OpenSketch, we use the arrows to show the pairs of false positives (+) and false negatives (X) with different size combination of Count-Min sketches and bitmaps calculated by our sketch manager. With the same 1MB memory, OpenSketch achieves 0.9% false positive rate and 0.3% false negative rate. Note that there are only 29 superspreaders on average during each measurement interval and thus the differences between OpenSketch and the streaming is only 1 superspreader at some intervals. OpenSketch can reach the same false positive and false negative rate as the streaming algorithm when it has 2.4MB memory.

Many measurement tasks can be implemented on top of OpenSketch platform with simple controller code and limited data plane resources: We have implemented *five* measurement tasks on top of OpenSketch (Table 2). The implementation is simple because we only configure existing building blocks and analyze the collected data to report the results. The configuration part takes about 10-25 lines of code and the analysis part takes at most 150 lines of code.

We also show the amount of data plane resources we need for each measurement task according to simulations and theoretical analysis. We only need 4-10 hash functions, less than 30 TCAM entries, and tens of megabytes SRAM to support most measurement tasks. For example, the flow size distribution task needs 3 TCAM entries for its multi-resolution classifier to index into one of three hash tables each of which covers a fraction ($\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$) of the flowspace. We can glue the building blocks together to count traffic in different ways. For example, we can count how many distinct senders contact a set of 200 destination address using a PCSA sketch and a Bloom Filter. We need 10 TCAM entries to store the destinations, each entry corresponds to a Bloom Filter for 20 addresses. For packets that pass the filter, PCSA counts the distinct senders (up to $\sim 2^{17}$) using 17 TCAM entries and 1KB of SRAM.

Resource allocation across measurement tasks: Figure 7 shows how OpenSketch allocates memory resources for two measurement tasks: heavy hitter and superspreader detection with different weight β, given the total memory size 4MB. With a lower β, we devote more resources to superspreaders to ensure its low error rate[5]. When we increase the β to 1 (heavy hitters have higher weights), heavy hitter detection gets more memory resources and thus higher accuracy.

6.2 Prototype evaluation

OpenSketch has no effect on data plane throughput: We deploy our NetFPGA based prototype into a Dell inspiron 530 machine (2 CPU cores and 2 GB DRAM). We connect 4 servers to the 4 Ethernet ports on NetFPGA. We first measure the throughput of OpenSketch switch. We set TCP flows across all four 1GE ports on NetFPGA. The OpenSketch prototype switch can achieve full 1GE throughput among four ports with different packet sizes (64, 512, and 1500 Bytes) without any packet losses. This is because the OpenSketch data plane pipeline does not interrupt the forwarding pipeline in the original switch, and the delay of each measurement pipeline component is smaller than packet incoming rate even for the 64Byte packets. As a result, the packets do not need to stay in the queue for measurement processing.

OpenSketch measurement performance is not affected by multiple hash functions, multiple wildcard rules, but is limited by counter updates. We setup a TCP flow between two servers across NetFPGA, and measure the processing delay of collecting statistics from single packet. We vary the number of hash functions from 1 to 8, and set the number of wildcard rules from 32 to 1024, respectively. The average delay is always 104 ns across all settings. This is because both hash functions and wildcard rules are implemented in parallel in hardware. However, the delay is affected by the number of counters we update for each packet. When we update 5 counters per packet in SRAM (which is the maximum number of updates a sketch may need), the processing delay increases to 200ns. This is because our SRAM can only be read and written sequentially. The performance can be improved with when fabrication of 6 to 8 read ports for an on-chip Random Access Memory is attainable with today's embedded memory technology [17].

[4] The paper [40] also proposes a two-level detection algorithm that only works better than one-level algorithm when there are lots of sources contacting only a few destinations. OpenSketch can also introduce another sampling layer as the two-level algorithm to further improve its accuracy.

[5] Note that here we are considering the error rate for superspreader counters. A 10-20% error rate is enough to ensure low false positive/negative rates.

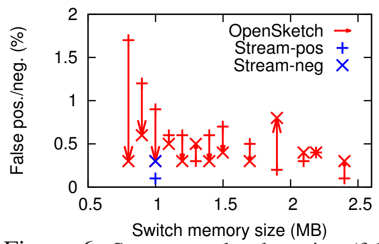


Figure 6: Superspreader detection (false positives (+), false negatives (x))

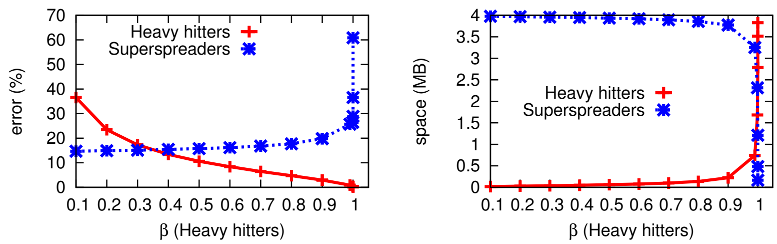


Figure 7: Resource allocation across Heavy hitters and Superspreaders (a) Error rate (b) Memory allocation

Meas. Tasks	Error (%)	Hash func	TCAM entries	SRAM size	Conf LOC	Ana. LOC
Heavy Hitters	0.05 10	3 for CountMin 5 for rev. sketch	0	94KB-600KB (89KB for rev.)	20	25
Superspreaders/ DDoS	0, 0 0.2, 0.2	3 for CountMin,1 for bitmap 5 for rev. sketch	0	0.9MB-1.5MB (0.5MB for rev.)	25	30
Detect traffic changes $>$ 0.5% of total changes	0.1-1	5 for 5-ary sketch, 5 for rev. sketch	0	3.2MB-32MB (82KB for rev.)	20	25
Flow size dist for 100K-2.5M flows	~1-2	1	3	300KB-7.5MB	20	150
Count traffic from$\leq$100K src to a set of 200 dst	0.1-1	1 for PCSA, 8 for Bloom Filter	17 for PCSA, 10-16 for B.F.	1KB for PCSA	10	5

Table 2: Implementing measurement tasks in OpenSketch (The numbers for the first two are based on our simulations. The numbers for the later three tasks are based on theoretical analysis in [36, 27, 23, 22]. The error rate is defined as: the relative error for heavy hitters, the false positive and false negative percentages for superspreaders, the relative error for traffic change detection, the weighted mean relative difference in simulation [27] for flow size distribution, and the overall false positive probability of the Bloom Filter for counting traffic (the distinct counter is configured for 10% relative error.))

7 Related Work

Programmable measurement architectures: In addition to NetFlow, there are other works that share our goal of building a configurable or programmable measurement architecture. ProgME [45] allows operators to specify *flowsets* and the switches count the packets in these flowsets. Gigascope [15] is a programmable packet monitor that supports queries on packet streams and automatically splits queries between the data plane and the control plane. In contrast, OpenSketch chooses sketches as the basis of the measurement architecture. Therefore, OpenSketch provides more compact data structures to store statistics with a provable memory-accuracy tradeoff, while supporting a wide range of measurement tasks.

The paper [37] extends NetFlow by using two sampling primitives (flow sampling and sample-and-hold) as the minimalist measurement support in switches, independent of the measurement tasks. Operators can only *passively* process the collected data for their measurement tasks. Other companies [3] build new hardware to provide more line-speed counters at switches. In contrast, OpenSketch allows operators to *proactively* configure different sketches and thus can best use the data plane with guaranteed accuracy for specific measurement tasks. OpenSketch also allows multiple measurement tasks to run at the same time.

Other flexible switch architecture: Software defined networks provide simple APIs at switches and allow the controller to program the switches based on the APIs. PLUG [11] provides flexible lookup modules for deploying routing protocols. OpenSketch shares the same goal of separating the data plane which processes packets, from the control plane that configures how to process the packets. However, existing proposals for software defined networks are not a good fit for measurement tasks. Recent work [16, 32] has recognized the problems of supporting different measurement tasks in OpenFlow [4], such as limited on-chip memory and large communication overhead between the controller and switches. Instead of incremental improvements on OpenFlow, we design a new software defined traffic measurement architecture that provides general and efficient measurement support at switches.

8 Conclusion

Like OpenFlow, which enables a simple, efficient way to control switches by separating the data and control plane, OpenSketch enables a simple and efficient way to collect measurement data. It uses data-plane measurement primitives based on commodity switches, and a flexible control plane so that operators can easily implement variable measurement algorithms. OpenSketch makes sketches more practical by bridging the gap between theoretical research in streaming algorithms and practical measurement requirements and constraints of switches, and makes sketches more flexible in supporting various measurement tasks.

References

[1] http://tools.ietf.org/html/draft-ietf-forces-protocol-22.

[2] http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[3] cpacket. http://www.cpacket.com/download/cPacket_cvu_family_overview_2011a.pdf/.

[4] OpenFlow switch. http://www.openflowswitch.org/.

[5] Opensketch code release. https://github.com/lavanyaj/opensketch.git.

[6] AL-FARES, RADHAKRISHNAN, M., RAGHAVAN, S., HUANG, B., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI* (2010).

[7] ALON, N., MATIAS, Y., AND SZEGEDY, M. The space complexity of approximating the frequency moments. In *STOC* (1996).

[8] BANDI, N., METWALLY, A., AGRAWAL, D., AND ABBADI, A. E. Fast data stream algorithms using associative memories. In *ACM SIGMOD* (2007).

[9] BAR-YOSSEF, Z., JAYRAM, T. S., KUMAR, R., SIVAKUMAR, D., AND TREVISAN, L. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques* (London, UK, UK, 2002), RANDOM '02, Springer-Verlag, pp. 1–10.

[10] BREMLER-BARR, A., HAY, D., AND KORAL, Y. CompactDFA: Generic state machine compression for scalable pattern matching. In *INFOCOM* (2010).

[11] CARLI, L. D., PAN, Y., KUMAR, A., ESTAN, C., AND SANKARALINGAM, K. PLUG: Flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *SIGCOMM* (2009).

[12] CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Finding hierarchical heavy hitters in streaming data. *ACM Transactions on Knowledge Discovery from Data* (Jan. 2008).

[13] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* (2005).

[14] CORMODE, G., AND MUTHUKRISHNAN, S. Summarizing and mining skewed data streams. In *In SIAM Conference on Data Mining* (2005).

[15] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A stream database for network applications. In *SIGMOD* (2003).

[16] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM* (2011).

[17] DIPERT, B. Special purpose SRAMs smooth the ride. *EDN* (1999).

[18] DUFFIELD, N., LUND, C., AND THORUP, M. Estimating flow distributions from sampled flow statistics. In *ACM SIGCOMM* (2003).

[19] ESTAN, C., KEYS, K., MOORE, D., AND VARGHESE, G. Building a better netflow. *ACM SIGCOMM* (2004).

[20] ESTAN, C., AND VARGHESE, G. New directions in traffic measurement and accounting. *ACM SIGCOMM* (2002).

[21] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high speed links. In *IMC* (2003).

[22] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci. 31*, 2 (Sept. 1985), 182–209.

[23] GOEL, A., AND GUPTA, P. Small subset queries and Bloom filters using ternary associative memories, with applications. In *ACM SIGMETRICS* (2010).

[24] HADJIELEFTHERIOU, M., BYERS, J. W., AND KOLLIOS, G. Robust sketching and aggregation of distributed data streams. Tech. rep., Boston University, 2005.

[25] HUANG, G., LALL, A., CHUAH, C.-N., AND XU, J. Uncovering global icebergs in distributed monitors. In *IEEE IWQoS* (2009).

[26] KOMPELLA, R., LEVCHENKO, K., SNOEREN, A., AND VARGHESE, G. Every microsecond counts: Tracking fine-grain latencies with a loss difference aggregator. In *ACM SIGCOMM* (2009).

[27] KUMAR, A., SUNG, M., XU, J., AND WANG, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *SIGMETRICS* (2004).

[28] MAI, J., CHUAH, C.-N., SRIDHARAN, A., YE, T., AND ZANG, H. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), IMC '06, ACM, pp. 165–176.

[29] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communication Review* (Apr. 2008).

[30] MEINERS, C. R., PATEL, J., NORIGE, E., TORNG, E., AND LIU, A. X. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *USENIX Security Symposium* (2010).

[31] MITZENMACHER, M., AND VADHAN, S. Why simple hash functions work: Exploiting the entropy in a data stream. In *SODA* (2008).

[32] MOGUL, J. C., AND CONGDON, P. Hey, you darned counters!: get off my asic! In *Proceedings of the first workshop on Hot topics in software defined networks* (New York, NY, USA, 2012), HotSDN '12, ACM, pp. 25–30.

[33] POPA, L., RATNASAMY, S., AND STOICA, I. Building extensible networks with rule-based forwarding. In *OSDI* (2010).

[34] PĂTRAŞCU, M., AND THORUP, M. The power of simple tabulation hashing. *J. ACM 59*, 3 (June 2012), 14:1–14:50.

[35] SANJUÀS-CUXART, J., BARLET-ROS, P., DUFFIELD, N., AND KOMPELLA, R. R. Sketching the delay: tracking temporally uncorrelated flow-level latencies. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 483–498.

[36] SCHWELLER, R., GUPTA, A., PARSONS, E., AND CHEN, Y. Reversible sketches for efficient and accurate change detection over network data streams. In *IMC* (2004).

[37] SEKAR, V., REITER, M. K., AND ZHANG, H. Revisiting the case for a minimalist approach for network flow monitoring. In *IMC* (2010).

[38] SHINDE, R., GOEL, A., GUPTA, P., AND DUTTA, D. Similarity search and locality sensitive hashing using TCAMs. In *SIGMOD* (2010).

[39] THORUP, M., AND ZHANG, Y. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM J. Comput. 41*, 2 (Apr. 2012), 293–331.

[40] VENKATARAMAN, S., SONG, D., GIBBONS, P. B., AND BLUM, A. New streaming algorithms for fast detection of superspreaders. In *Network and Distributed System Security Symposium* (2005).

[41] WALSWORTH, C., ABEN, E., KC CLAFFY, AND ANDERSEN, D. The CAIDA Anonymized 2009 Internet Traces - Sep. 17 2009. `http://www.caida.org/data/passive/passive_2009_dataset.xml`.

[42] WANG, M., LI, B., AND LI, Z. sflow: Towards resource-efficient and agile service federation in service overlay networks. *Distributed Computing Systems, International Conference on 0* (2004), 628–635.

[43] WHANG, K.-Y., VANDER-ZANDEN, B. T., AND TAYLOR, H. M. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst. 15*, 2 (June 1990), 208–229.

[44] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with opensketch. Tech. rep., USC Technical Report, 2013.

[45] YUAN, L., CHUAH, C.-N., AND MOHAPATRA, P. ProgME: Towards programmable network measurement. In *ACM SIGCOMM* (2007).

[46] ZHANG, Y., SINGH, S., SEN, S., DUFFIELD, N., AND LUND, C. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In *IMC* (2004).

[47] ZHAO, Q., XU, J., AND LIU, Z. Design of a novel statistics counter architecture with optimal space and time efficiency. In *ACM SIGMETRICS* (2006).

[48] ZSEBY, T., ET AL. Sampling and filtering techniques for IP packet selection. `http://www.ietf.org/internet-drafts/draft-ietf-psamp-sample-tech-07.txt`, July 2005. –Work in progress.