SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders

Lu Tang[1], Qun Huang[2], and Patrick P. C. Lee[1]
[1]Department of Computer Science and Engineering, The Chinese University of Hong Kong
[2]State Key Lab of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences

***Abstract*—Superspreaders (i.e., hosts with numerous distinct connections) remain severe threats to production networks. How to accurately detect superspreaders in real-time at scale remains a non-trivial yet challenging issue. We present SpreadSketch, an invertible sketch data structure for network-wide superspreader detection with the theoretical guarantees on memory space, performance, and accuracy. SpreadSketch tracks candidate superspreaders and embeds estimated fan-outs in binary hash strings inside small and static memory space, such that multiple SpreadSketch instances can be merged to provide a network-wide measurement view for recovering superspreaders and their estimated fan-outs. We present formal theoretical analysis on SpreadSketch in terms of space and time complexities as well as error bounds. Trace-driven evaluation shows that SpreadSketch achieves higher accuracy and performance over state-of-the-art sketches. Furthermore, we prototype SpreadSketch in P4 and show its feasible deployment in commodity hardware switches.**

I. Introduction

Identifying *superspreaders* (i.e., hosts with a large number of distinct connections) in real-time is crucial in various network management tasks, including hot-spot localization in peer-to-peer networks [32] and detection of malicious attacks (e.g., DDoS attacks [11], port scanning [8], and worm propagation [33]). For example, superspreaders may refer to the infected hosts that connect to many other hosts for worm propagation [33], or the servers overwhelmed by a botnet of zombie hosts under DDoS attacks [11]. Despite many efforts in superspreader detection over decades, superspreaders (e.g., DDoS attacks) remain widespread in modern production networks [11].

One challenge of superspreader detection is that superspreaders are *distributed* by nature, as their myriad connections may span the entire network. A superspreader may appear with only a small number of connections at a measurement point (e.g., end-host or switch), but its aggregation across multiple measurement points may have a significant number of connections. Thus, it is essential to detect superspreaders *in real-time at scale*, based on a *network-wide* view aggregated from multiple measurement points [27]. A simple network-wide detection approach is to maintain complete per-flow states [28], [30], yet the memory consumption becomes prohibitive in large-scale networks when a surge of active flows appear in short timescales (e.g., under DDoS attacks).

The necessity of network-wide superspreader detection motivates us to explore compact data structures that enable memory-efficient measurement with provable accuracy guarantees. In particular, we focus on *invertible sketches*, the summary data structures that count the number of distinct connections using fixed-size memory footprints with bounded errors, while supporting the fast recovery of all superspreaders from the data structures only (i.e., invertibility). Invertible sketches are particularly critical for network-wide measurement, in which we can aggregate multiple invertible sketches (which retain the superspreader information) from various measurement points across the network in order to recover all superspreaders in a network-wide view. However, existing invertible sketches for superspreader detection (e.g., [7], [14], [24], [26], [36], [42]), often face different performance challenges. They either incur high memory access overhead that slows down packet processing, or incur high computational overhead that delays the recovery of superspreaders (Section II-C). Thus, we pose the following question: *Can we design an invertible sketch for network-wide superspreader detection that simultaneously achieves (i) memory efficiency, (ii) high packet processing and superspreader detection performance, as well as (iii) high detection accuracy?*

We present SpreadSketch, an invertible sketch for network-wide superspreader detection with the theoretical guarantees on memory space, performance, and accuracy. SpreadSketch maps each observed connection (e.g., a source-destination pair) to a binary hash string that estimates the fan-out (i.e., number of distinct connections) of the host based on the length of the most significant zero bits (as in Probabilistic Counting [13]). It tracks candidate superspreaders in a fixed-size table of buckets, such that multiple SpreadSketch instances can be merged to provide a network-wide view for recovering all superspreaders and their estimated fan-outs. In particular, SpreadSketch maintains its sketch with small and static memory allocation and requires only simple computations (e.g., multiplications, shifts, and hashing). Such features not only improve packet processing performance (without dynamic memory allocation), but also enable SpreadSketch to be feasibly deployed in both software and hardware and serve as a building block in current network-wide measurement systems [17], [18], [23], [27], [29], [39], [42]. In summary, this paper makes the following contributions:

- We design SpreadSketch, a new invertible sketch data structure for network-wide superspreader detection with memory space, performance, and accuracy guarantees.
- We present formal theoretical analysis on SpreadSketch, including its space complexity, update and detection time complexities, as well as error bounds on superspreader detection and fan-out estimation.

Qun Huang is now affiliated with Peking University

- We show via trace-driven evaluation that SpreadSketch achieves higher detection accuracy, higher update throughput, and lower detection time than state-of-the-art sketches.
- We prototype SpreadSketch in P4 [3] and compile it in the Barefoot Tofino chipset [1]. We present microbenchmark results on SpreadSketch to justify its feasible deployment in commodity hardware switches.

The source code of our SpreadSketch prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/spreadsketch**.

II. Background and Related Work

A. Superspreader Detection

We consider the processing of a stream of packets, each of which is denoted as a source-destination pair (x, y) and is allowed to be processed only once. The source x can refer to any combination of packet header fields that identify the source of the packet, such as the source IP address, or the pair of source IP address and port; similar definitions apply to the destination y. We assume that both source x and destination y are the unique *keys* that are drawn from a key space represented as an integer domain $[n] = \{0, 1, \cdots, n-1\}$; in other words, a key can be represented in $\lceil \log_2 n \rceil$ bits. Note that each (x, y) can appear multiple times in a packet stream.

We formulate the superspreader detection problem as follows. We conduct superspreader detection at regular time intervals called *epochs*. We define the *fan-out* of a source as the number of distinct destinations to which the source connects over an epoch. We say that a source is a *superspreader* if its fan-out exceeds a pre-defined threshold. Formally, let ϕ $(0 < \phi < 1)$ be a pre-defined fractional threshold for distinguishing superspreaders from a packet stream. A source x is a superspreader if $S(x) \geq \phi\mathcal{S}$, where $S(x)$ is the fan-out of x and $\mathcal{S}$ is the total fan-out of all sources appearing in the epoch. In practice, we can obtain $\mathcal{S}$ with distinct counting (e.g., [9], [12], [13], [37]) that accurately estimates the number of distinct source-destination pairs with small memory space. We assume that both the epoch length and the threshold are manually configured by network administrators.

We can also symmetrically estimate the number of distinct sources with which a destination is connected over an epoch, and a superspreader refers to a destination that is connected by many distinct sources (e.g., under DDoS attacks). Without loss of generality, we use the term "superspreader" to refer to superspreader sources throughout the paper.

B. Design Requirements

Given the resource constraints of tracking all active flows in large-scale networks (Section I), our primary goal is to design a practical *sketch* data structure for superspreader detection, with the following design requirements.

- **Invertibility:** The sketch itself can readily return all superspreaders and their fan-outs at the end of an epoch.
- **Network-wide detection:** We can deploy the sketch at multiple measurement points (e.g., end-hosts or switches) across the network to perform network-wide superspreader detection. Specifically, we can aggregate the local results at multiple measurement points as if all network traffic was measured at a big measurement point [27].
- **Small and static memory usage:** The sketch incurs small memory footprints, which are essential for the deployment in both software and hardware [18]. Also, its memory resources can be statically allocated in advance to avoid dynamic memory management overhead [34].
- **High detection accuracy:** The sketch achieves high detection accuracy with small memory footprints and provable error bounds. Note that the accuracy guarantees should be preserved even in the worst-case scenarios, such as malicious attacks or traffic bursts.
- **Fast updates and detection:** The sketch supports high-speed per-packet updates. For example, a fully utilized 10 Gb/s link with 64-byte packets implies that the sketch can be updated with at least 14.88 million packets per second. Also, the sketch should detect and return all superspreaders in real-time to quickly react to any possibly ongoing attacks.

C. Limitations of Existing Approaches

While superspreader detection has been extensively studied in the literature, we argue that no existing approaches can address all design requirements in Section II-B.

Per-source tracking. Traditional intrusion detection systems (e.g., Snort [28] and FlowScan [30]) maintain all active connections for each source to identify any port scans or DDoS attacks. To improve memory efficiency, Estan et al. [9] propose a small triggered bitmap that counts only the sources with high fan-outs. However, per-source tracking incurs tremendous resource usage, especially for high-speed links that contain numerous active flows.

Sampling. To limit packet processing overhead, *hash-based flow sampling* [5], [19], [35] is proposed to monitor (deterministically) only a fraction of flows whose hashed flow IDs are less than some pre-specified threshold (i.e., the sampling probability). Thus, the superspreaders are likely to be sampled as they have high fan-outs. However, sampling inherently has low detection accuracy in short timescales [23]. Also, some approaches [5], [19] maintain sampled flows in chained hash tables, which have high memory access overhead over the linked lists of buckets.

Streaming. Various streaming algorithms are designed for superspreader detection. Zhao et al. [43] combine sampling (which filters hosts with small flow counts) and streaming (which estimates the fan-out of each sampled source). Some approaches [22], [41] estimate the fan-outs of sources in tight memory space by sharing counter bits among sources. The above solutions design compact data structures that fit in fast memory (e.g., SRAM) for fan-out estimation. However, such data structures are *non-invertible* and cannot directly return all superspreaders from only the data structures in fast memory.

Sketches. Sketches are summary data structures that track all packets in fixed-size memory footprints. Several sketches in the literature aim for superspreader detection and also address invertibility by design.

Distinct-Count Sketch [14] extends the idea of Probabilistic Counting [13] to track the fan-outs. To track the list of

superspreaders, it also maintains a counter for every bit of a source-destination pair, thereby resulting in high memory usage and access overhead.

Some approaches encode the superspreader information into a sketch, and later enumerate the entire source key space to recover the candidate superspreaders. Connection Degree Sketch [36] reconstructs host addresses associated with large fan-outs based on the Chinese Remainder Theorem. Vector Bloom Filter [24] improves update efficiency via bit-extraction hashing, which extracts bits directly from the source ID. However, the computational overhead of recovering superspreaders is significant for very large key space.

Several studies propose *cascaded sketches*, whose idea is to combine existing invertible frequency-based sketches (i.e., the sketches that return high-frequency keys) and distinct counting, so as to recover all source keys with high fan-outs. Count-Min-Heap [7] extends Count-Min sketch [6] with an external heap for tracking superspreaders and associates each bucket with a distinct counter (e.g., [9], [12], [13], [37]). OpenSketch [42] combines Reversible Sketch [31] with bitmap algorithms [9], while Liu et al. [26] combine Fast Sketch [25] with the optimal distinct counter [20], for superspreader detection. However, existing invertible frequency-based sketches generally have high processing overhead [34]: the external heap of the Count-Min-Heap incurs high memory access overhead for heap updates, while Reversible Sketch and Fast Sketch incur an update overhead that grows linearly with the key size.

D. Other Related Work

Detecting heavy hitters. Recent studies (e.g., [4], [15], [34]) focus on detecting heavy hitters (i.e., the sources whose frequencies exceed a pre-defined threshold). Superspreader detection can be viewed as a special case of heavy hitter detection by identifying the sources with many distinct connections. However, existing heavy hitter detection solutions cannot be directly applied to superspreader detection as they cannot distinguish duplicate connections in packet streams.

General network-wide measurement. Network-wide measurement systems [17], [18], [23], [27], [29], [39], [42] propose unified frameworks for general measurement tasks, and some of them [17], [27], [29], [42] also address superspreader detection. Our proposed sketch design (Section III) can be a building block of the above network-wide measurement systems.

Stealthy spreaders. Some variants of the superspreader detection problem are covered in the literature. Yoon et al. [40] design a random aging streaming filter to find stealthy spreaders that send low-rate malicious packets. Xiao et al. [38] and Zhou et al. [44] study the estimation of persistent fan-outs over a number of epochs. Huang et al. [16] further address the estimation of the k-persistent fan-outs that appear in at least k out of a fixed number of epochs. The above problems are different with ours, and we pose them as future work.

III. SpreadSketch Design

SpreadSketch is a novel sketch data structure for superspreader detection that addresses all design requirements in Section II-B. It takes a clean-slate approach that incorporates invertibility and network-wide detection by design, while providing the theoretical guarantees on memory space, performance, and accuracy (Section IV).

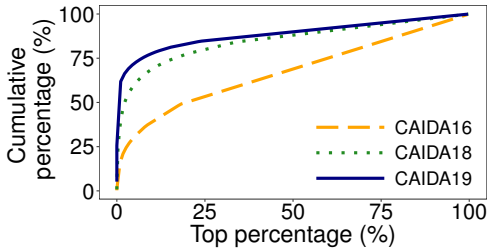


Fig. 1. Cumulative fan-out ratios of the top-percentages of sources (i.e., the sum of fan-outs of the top-percentage of sources over the total fan-outs of all sources) for three real-world IP traces.

A. Main Idea

SpreadSketch is a non-trivial extension of the classical *Count-Min Sketch* [6]. Count-Min Sketch is initialized with multiple rows of *buckets*, each of which is associated with a general integer counter. For each item in a stream, Count-Min Sketch hashes the item key into a bucket in each row and increments the associated counter by the item value. It provides an estimate for the value sum of an item key using the minimum counter value of all buckets hashed by the item key.

SpreadSketch augments Count-Min Sketch by associating each bucket with a *distinct counter*, a small fixed-size data structure that counts the distinct items of a stream (e.g., [9], [12], [13], [37]). Also, each bucket of SpreadSketch tracks the key of a candidate superspreader that is estimated to have the most fan-outs among all sources that are hashed to the bucket. SpreadSketch's design is motivated by two observations.

Highly skewed fan-out distributions. Fan-out distributions in practice are highly skewed, in which a small fraction of sources have significantly higher fan-outs than the remaining majority of sources. Figure 1 plots the cumulative fan-out ratios of the top-percentage of sources (sorted by fan-outs in descending order) for three real-world IP traces (see Section V for trace details). We observe different degrees of skewness of different traces. For example, the top 1% of sources account for over 60% of total fan-outs in the most skewed CAIDA19; the top 10% of sources account for over 67% of total fan-outs in the moderately skewed CAIDA18; the top 10% of sources account for over 38% of total fan-outs for the least skewed CAIDA16. Thus, it is highly likely that the fan-out of each bucket of SpreadSketch is dominated by at most one superspreader, while we use multiple buckets to mitigate the hash collisions of multiple superspreaders into the same bucket.

Accurate fan-out estimation via hash strings. Inspired by Probabilistic Counting [13], we can construct a binary *hash string* of 0's and 1's by hashing each source-destination pair. We then use the hash string to estimate the fan-out of a source. Suppose that the hash string is uniformly generated in the form of 0^l1*, where l (called the *level*) denotes the length of the most significant 0-bits and $*$ represents arbitrary bits. Then on average $1/2^{l+1}$ of distinct pairs have the pattern 0^l1*. In other

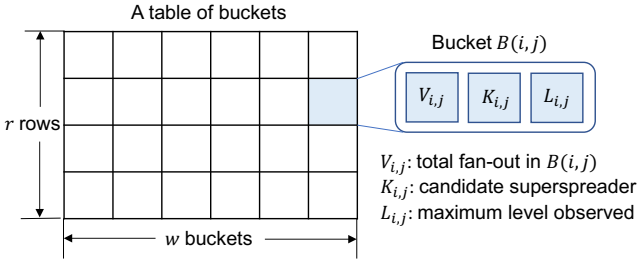


Fig. 2. Data structure of SpreadSketch.

words, the level l provides a rough estimation of the number of distinct pairs. Given that each bucket is likely hashed by at most one superspreader (see above), it can track the source with the largest level as the candidate superspreader.

Also, we can combine the hash strings of a particular source from multiple measurement points and find the candidate (network-wide) superspreader in each bucket. Even though a superspreader has a small fan-out at each single measurement point, as long as its total fan-out dominates its hashed buckets, we can still identify it via its combined hash string with a high probability. We show how we leverage this property for tracking candidate superspreaders in SpreadSketch in network-wide detection (Section III-F).

B. Data Structure

Figure 2 shows the data structure of SpreadSketch, which comprises r rows with w buckets each. Let $B(i,j)$ be the bucket at the i-th row and the j-th column, where $1 \le i \le r$ and $1 \le j \le w$. Each bucket $B(i,j)$ consists of three fields: (i) $V_{i,j}$, which is the distinct counter that counts the sum of fan-outs of all sources hashed to the bucket (let $|V_{i,j}|$ denote the value stored in $V_{i,j}$); (ii) $K_{i,j}$, which stores the key of the current candidate source that has the maximum level in the bucket; and (iii) $L_{i,j}$, which stores the current maximum level observed in the bucket. Note that we can pre-allocate static memory space for SpreadSketch in advance before the measurement starts.

In addition, SpreadSketch is associated with two sets of hash functions: (i) r pairwise-independent hash functions, denoted by $h_1, h_2, \cdots, h_r$, such that h_i $(1 \le i \le r)$ hashes a source key into one of the w buckets in row i; and (ii) the global hash function h^*, which transforms each source-destination pair into a hash string that closely resembles truly uniform independent bits. Note that h^* can be realized via many practical hash schemes (e.g., standard multiplicative hashing) whose outputs are indistinguishable from truly random bits [12], [21].

C. Basic Operations

SpreadSketch supports two basic operations (Figure 3): (i) *Update*, which updates a source-destination pair (x,y) into the sketch; and (ii) *Query*, which returns the estimated fan-out of an input source key x.

The Update operation (Lines 1-9 of Figure 3) is invoked for each arrival of (x,y) in a packet stream. We initialize the variables of all buckets of SpreadSketch to zeros. Upon the arrival of (x,y), we first compute the hash string of (x,y) via

```
1: procedure UPDATE(x, y)
2:     l ← length of most significant 0-bits of h*(x, y)
3:     for i = 1 to r do
4:         COUNT(V_{i,h_i(x)}, x, y)
5:         if L_{i,h_i(x)} ≤ l then
6:             (K_{i,h_i(x)}, L_{i,h_i(x)}) ← (x, l)
7:         end if
8:     end for
9: end procedure
10: procedure QUERY(x)
11:     return Ŝ(x) ← min_{1≤i≤r}{|V_{i,h_i(x)}|}
12: end procedure
13: procedure COUNT(V, x, y)
14:     l ← length of most significant 0-bits of h*(x, y)
15:     if l < c − 1 then
16:         p ← h*(x, y) mod b
17:         V[l][p] ← 1
18:     else
19:         p ← h*(x, y) mod b′
20:         V[c − 1][p] ← 1
21:     end if
22: end procedure
23: procedure MERGE(q)
24:     for i = 1 to r do
25:         for j = 1 to w do
26:             V_{i,j} ← V^1_{i,j} ∪ V^2_{i,j} ··· ∪ V^q_{i,j}
27:             K_{i,j} ← K^{k*}_{i,j}, where k* = arg max_{1≤k≤q}{L^k_{i,j}}
28:             L_{i,j} = max_{1≤k≤q}{L^k_{i,j}}
29:         end for
30:     end for
31: end procedure
```

Fig. 3. Main operations of SpreadSketch.

the hash function h^* and obtain the level l of the hash string (i.e., the length of the most significant 0-bits). For each row i $(1 \le i \le r)$, we hash the source x into the bucket $B(i, h_i(x))$ and increment the distinct counter $V_{i,h_i(x)}$. We also compare l with the current maximum level $L_{i,h_i(x)}$: if $L_{i,h_i(x)} \le l$, then we replace $K_{i,h_i(x)}$ with x and update $L_{i,h_i(x)}$ to l, meaning that x is now the source with the maximum level among all sources hashed to the bucket.

The Query operation (Lines 10-12 of Figure 3) is invoked for a given input source x. We extract the value of each distinct counter associated with x for $1 \le i \le r$ (denoted by $|V_{i,h_i(x)}|$). We return the minimum value of all the distinct counters as the estimated fan-out of x (denoted by $\hat{S}(x)$).

D. Distinct Counters

Both Update and Query operations of SpreadSketch depend on the choice of the distinct counter (denoted by V) associated with each bucket. In this paper, we choose the *multiresolution bitmap* [9], which supports multiset operations for network-wide detection (Section III-F) and can be readily implemented in hardware (Section V).

The Update operation calls the Count operation (Lines 13-22 of Figure 3) for distinct counting, which we realize as follows. We construct the distinct counter of each bucket as c bitmaps $V[0], V[1], \cdots, V[c-1]$, where the first $c-1$ bitmaps $V[0], V[1], \cdots, V[c-2]$ have b bits each and are associated with the hash strings $0^0 1*, 0^1 1*, \cdots, 0^{c-2} 1*$, respectively,

while $V[c-1]$ has b' bits and is associated with the hash strings that have at least $c-1$ most significant 0-bits; note that c, b, and b' are configurable parameters (see details below). Given the hash string $h^*(x,y)$, we map it to the corresponding bitmap according to the number of most significant 0-bits, and set the p-th bit to one, where p is the hash string modulo the bitmap size. Based on the bitmap configuration, we expect that half of the distinct items are mapped to $V[0]$, a quarter of the distinct items are mapped to $V[1]$, and so on. To estimate the distinct count of a multiresolution bitmap, we add all the distinct counts of all bitmaps and multiply the sum with some sampling factor [9].

The Query operation returns the minimum value of multiple distinct counters associated with a source. We can estimate the minimum value by combining multiple multiresolution bitmaps via the bitwise AND operation and obtaining the distinct count estimate of the combined bitmap.

We configure c, b, and b' according to some pre-specified relative error σ ($0 < \sigma < 1$) and the maximum possible distinct count C [10]. Specifically, we fix $b = 0.6367/\sigma^2$, and initialize $b' = 2b$ and $c = 2 + \lceil \log_2(C/2.6744b) \rceil$. We fine-tune both b' and c for the minimum memory usage subject to the inputs C and σ via the *ComputeConfiguration* algorithm. We refer readers to [10] for details.

E. Identification of Superspreaders

To recover all superspreaders, we check all $r \times w$ buckets of SpreadSketch at the end of each epoch. For each bucket $B(i,j)$ ($1 \le i \le r$, $1 \le j \le w$), if the value of $V_{i,j}$ exceeds the pre-specified threshold, then we call the Query operation on the candidate source in $K_{i,j}$ to estimate its fan-out. If the estimated fan-out also exceeds the pre-specified threshold, we report the candidate source as a superspreader.

F. Network-Wide Superspreader Detection

We can deploy SpreadSketch at multiple measurement points in parallel to support network-wide superspreader detection. Suppose that there are q measurement points and a centralized controller. Each measurement point runs an instance of SpreadSketch, where all instances share the same set of parameters (e.g., r, w, and hash functions). At the end of each epoch, each measurement point sends its sketch data structure to the controller, which then merges all received sketches (via a Merge operation) and recovers superspreaders based on the merged sketch. Note that we do not make any assumptions on the selection of measurement points or the traffic distributions among the measurement points.

We elaborate the Merge operation as follows (Lines 23-31 of Figure 3). Let $B^k(i,j) = (V^k_{i,j}, K^k_{i,j}, L^k_{i,j})$ be the bucket with index (i,j) at the k-th measurement point, where $1 \le i \le r$, $1 \le j \le w$, and $1 \le k \le q$. Upon receiving all q sketches, the controller constructs a *merged sketch* whose bucket $B(i,j)$ is formed by all $B^k(i,j)$'s: (i) it sets $V_{i,j}$ as the union of all $V^k_{i,j}$'s (for the multiresolution bitmap [9], the union is equivalent to the bitwise OR operation); (ii) it sets $K^k_{i,j}$ as the candidate superspreader that has the maximum level among all $K^k_{i,j}$; and (iii) it sets $L_{i,j}$ as the maximum value of all $L^k_{i,j}$'s.

After the Merge operation, the controller performs superspreader detection on the merged sketch as in Section III-E. In essence, the merged sketch provides a network-wide view as if all traffic were measured at a big measurement point.

IV. Theoretical Analysis

We present theoretical analysis on SpreadSketch in memory space, performance, and accuracy. We configure SpreadSketch with three parameters: ϵ, δ, and σ ($0 < \epsilon, \delta, \sigma < 1$), where ϵ and δ are the approximation parameter and the error probability for the sketch configuration, respectively, and σ is the error factor for the distinct counter configuration. We set $r = \log \frac{1}{\delta}$ and $w = \frac{2}{\epsilon}$, where the logarithm base is 2. Our analysis also assumes $\epsilon \le \frac{\phi}{4}$ to provide provable error bounds. Given σ, we can derive the minimum memory space m (in bits) for a distinct counter (Section III-D).

A. Space and Time Complexities

Theorem 1 shows the complexities of memory space, update time, and detection time of SpreadSketch.

Theorem 1. *The memory space is* $O(\frac{m+\log n+\log\log n}{\epsilon} \log \frac{1}{\delta})$. *The per-packet update time is* $O(\log \frac{1}{\delta})$, *while the detection time of returning all superspreaders is* $O(\frac{1}{\epsilon} \log^2 \frac{1}{\delta})$.

Proof. SpreadSketch has rw buckets, each of which holds an m-bit distinct counter ($V_{i,j}$), a $\log n$-bit candidate source key ($K_{i,j}$), and a $\log\log n$-bit level counter ($L_{i,j}$). Thus, the memory space is $O(rw(m + \log n + \log\log n)) = O(\frac{m+\log n+\log\log n}{\epsilon} \log \frac{1}{\delta})$.

Each per-packet update takes one hash operation for calculating the level, and then accesses $\log \frac{1}{\delta}$ buckets to update distinct counters. Thus, it takes $O(\log \frac{1}{\delta} + 1) = O(\log \frac{1}{\delta})$ time.

SpreadSketch checks all buckets to identify all superspreaders and estimate their fan-outs. It traces at most rw superspreaders, each of which checks r buckets for its fan-out estimation. The detection time is $O(r^2 w) = O(\frac{1}{w} \log^2 \frac{1}{\delta})$. □

Note that our proof of Theorem 1 assumes that each distinct counter has $O(1)$ time complexities, including adding an item to the distinct counter and estimating the distinct count. This assumption holds for the multiresolution bitmap [9] that we use and other distinct counters [20], [37].

B. Accuracy for the Estimated Fan-Out

Theorem 2 shows the lower and upper bounds of the estimated fan-out $\hat{S}(x)$ of a source x from the Query operation. We bound $\hat{S}(x)$ with respect to $S(x)$ (i.e., the true fan-out of x) and $\mathcal{S}$ (i.e., the total fan-out of all sources) (Section II-A).

Theorem 2. *For any source* x, $\hat{S}(x) \ge (1-\sigma)S(x)$; *with a probability at least* $1-\delta$, $\hat{S}(x) \le (1+\sigma)(S(x) + \epsilon\mathcal{S})$.

Proof. For the lower bound, each distinct counter associated with x is updated by at least $S(x)$ times. Given an error factor σ, the estimate returned by each distinct counter is at least $(1-\sigma)S(x)$. As $\hat{S}(x)$ takes the minimum estimate of all distinct counters, we have $\hat{S}(x) \ge (1-\sigma)S(x)$.

For the upper bound, let R_i be the sum of fan-outs of all sources excluding x in the bucket $B(i, h_i(x))$ hashed by x in row i, where $1 \le i \le r$. The expectation of R_i, denoted by $E[R_i]$, is given by $E[\sum_{z \neq x, h_i(z)=h_i(x)} S(z)] \le \frac{\mathcal{S}-S(x)}{w} \le \frac{\epsilon\mathcal{S}}{2}$, due to the pairwise independence of h_i and the linearity of expectation. By Markov's inequality,

$$\Pr[R_i \ge \epsilon\mathcal{S}] \le \tfrac{1}{2}. \quad (1)$$

Given an error factor σ, we have $\hat{S}(x) \le (1+\sigma)(R_i+S(x))$ for each row i. Thus, $\Pr[\hat{S}(x) \le (1+\sigma)(S(x)+\epsilon\mathcal{S})]$

$= 1 - \Pr[\hat{S}(x) - (1+\sigma)S(x) \ge (1+\sigma)\epsilon\mathcal{S}]$

$\ge 1 - \Pr[(1+\sigma)(R_i+S(x)) - (1+\sigma)S(x) \ge (1+\sigma)\epsilon\mathcal{S}, \forall_i]$

$= 1 - \Pr[R_i \ge \epsilon\mathcal{S}, \forall_i] \ge 1 - (\frac{1}{2})^r = 1 - \delta.$

The theorem follows. □

C. Accuracy for Superspreader Detection

We first analyze the likelihood that a superspreader is tracked by SpreadSketch. We then study the false positive and false negative rates of SpreadSketch.

Lemma 1. *A superspreader x is stored in one of its hashed buckets with a probability at least $1-\delta$.*

Proof. We first describe the idea of our proof. Consider the bucket $B(i, h_i(x))$ hashed by x in row i, where $1 \le i \le r$. We partition the distinct source-destination pairs hashed to the bucket into two groups. The first group contains $S(x)$ distinct pairs sharing the same source x, and the second group contains the remaining R_i distinct pairs. Let $M_{S(x)}$ and M_{R_i} be the maximum level values of the two groups, respectively. If $M_{S(x)} > M_{R_i}$, x is stored in $K_{i,h_i(x)}$. Our idea is to show that if x is a superspreader, then $M_{S(x)} \le M_{R_i}$ with a very small probability. Let A denote the event $M_{S(x)} \le M_{R_i}$. Then our problem is to prove that the probability of A, $\Pr[A]$, is at most δ. We prove this in several steps.

Step (i): Deriving the close-form expression of $\Pr[A]$. Let X_t, where $1 \le t \le S(x)$, be a random variable that denotes the length of the most significant 0-bits of a hash string for the t-th distinct pair of the source x. X_t follows a geometric distribution with the parameter $\frac{1}{2}$. Thus, we have $\Pr[X_t \le l] = 1 - \frac{1}{2^{l+1}}$. Recall that $M_{S(x)}$ is the maximum level value among the mutually independent random variables $\{X_t\}_{1 \le t \le S(x)}$, we have $\Pr[M_{S(x)} \le l] = \prod_{1 \le t \le S(x)} \Pr[X_t \le l] = (1 - \frac{1}{2^{l+1}})^{S(x)}$. Similarly, we obtain $\Pr[M_{R_i} \le l] = (1 - \frac{1}{2^{l+1}})^{R_i}$. Thus,

$$\begin{aligned}
\Pr[A] &= \textstyle\sum_{l \ge 0} \Pr[M_{R_i} = l \text{ and } M_{S(x)} \le l] \\
&= \textstyle\sum_{l \ge 0} \Pr[M_{R_i} = l] \Pr[M_{S(x)} \le l] \\
&= \textstyle\sum_{l \ge 0} (\Pr[M_{R_i} \le l] - \Pr[M_{R_i} \le l-1]) \Pr[M_{S(x)} \le l] \\
&= \textstyle\sum_{l \ge 0} ((1 - \frac{1}{2^{l+1}})^{R_i} - (1 - \frac{1}{2^l})^{R_i})(1 - \frac{1}{2^{l+1}})^{S(x)}. \quad (2)
\end{aligned}$$

Step (ii): Analyzing the upper bound of $\Pr[A]$. Let $R_i = \lambda S(x)$ for some constant $\lambda > 0$. Based on Equation (2), we rewrite $\Pr[A]$ as a function of R_i and λ, denoted by $F(R_i; \lambda)$:

$$F(R_i; \lambda) = \textstyle\sum_{l \ge 0} ((1 - \frac{1}{2^{l+1}})^{R_i} - (1 - \frac{1}{2^l})^{R_i})(1 - \frac{1}{2^{l+1}})^{R_i/\lambda}.$$

We can validate that $F(R_i; \lambda)$ is a decreasing function of R_i and an increasing function of λ.

This helps us simplify $\Pr[A]$ and obtain a rough upper bound. We split R_i into equal-length ranges $I_l = [\frac{lS(x)}{4}, \frac{(l+1)S(x)}{4}]$ for integer $l \ge 0$. Let $P(l) = \Pr[R_i \in I_l]$ for $l \ge 0$. Thus,

$$\begin{aligned}
\Pr[A] &= \textstyle\sum_{l \ge 0} \Pr[A \mid R_i \in I_l] P(l) \\
&< \textstyle\sum_{l \ge 0} \Pr[A \mid \lambda = \frac{(l+1)}{4}] P(l) \\
&< F(R_i; \tfrac{1}{4}) P(0) + F(R_i; \tfrac{1}{2}) P(1) + \Pr[R_i \ge \tfrac{S(x)}{2}]. \quad (3)
\end{aligned}$$

To obtain the exact upper bound of $\Pr[A]$, we maximize the right hand side of Inequality (3) by configuring its variables. We first set $P(0)$, $P(1)$, and $\Pr[R_i \ge \frac{S(x)}{2}]$, given the condition that $P(0)+P(1)+\Pr[R_i \ge \frac{S(x)}{2}] = 1$. If x is a superspreader, $S(x) \ge \phi\mathcal{S}$. By the assumption $\epsilon \le \frac{\phi}{4}$ and Inequality (1), we have $\Pr[R_i \ge \frac{S(x)}{4}] \le \Pr[R_i \ge \frac{\phi\mathcal{S}}{4}] \le \Pr[R_i \ge \epsilon\mathcal{S}] \le \frac{1}{2}$; in other words, $P(0) \ge \frac{1}{2}$. Similarly, $\Pr[R_i \ge \frac{S(x)}{2}] \le \frac{1}{4}$.

Since $F(R_i; \lambda)$ increases with λ, $F(R_i; \frac{1}{4}) < F(R_i; \frac{1}{2}) < 1$. The right hand side of Inequality (3) is maximized when $P(0) = \frac{1}{2}$, $P(1) = \frac{1}{4}$, and $\Pr[R_i \ge \frac{S(x)}{2}] = \frac{1}{4}$. Thus,

$$\Pr[A] < F(R_i; \tfrac{1}{4}) \times \tfrac{1}{2} + F(R_i; \tfrac{1}{2}) \times \tfrac{1}{4} + \tfrac{1}{4}.$$

Step (iii): Quantifying the upper-bound of $\Pr[A]$. Here, we configure some practical values of $S(x)$ to quantify the terms $F(R_i; \frac{1}{4})$ and $F(R_i; \frac{1}{2})$. For example, suppose that $S(x) > 10$. We have $\Pr[A] < 0.269 \times \frac{1}{2} + 0.422 \times \frac{1}{4} + \frac{1}{4} = 0.49 < \frac{1}{2}$.

By considering all r rows, we show that the probability that a superspreader x is not tracked by all r hashed buckets is $\Pr[M_{R_i} \ge M_{S(x)}, \forall_{1 \le i \le r}] < \frac{1}{2^r} = \delta$. The theorem follows. □

Theorems 3 and 4 bound the false negative and false positive rates of SpreadSketch, respectively.

Theorem 3. *For source x with $S(x) \ge \frac{\phi\mathcal{S}}{1-\sigma}$, SpreadSketch reports x as a superspreader with a probability at least $1-\delta$.*

Proof. By Theorem 2, $\hat{S}(x) \ge (1-\sigma)S(x) \ge \phi\mathcal{S}$. Then x is not reported as a superspreader if and only if it is not stored in any of its hashed buckets. By Lemma 1, this happens with a probability at most δ. Thus, x is reported as a superspreader with a probability at least $1-\delta$. □

Theorem 4. *For source x with $S(x) \le \frac{\epsilon\mathcal{S}}{1+\sigma}$, SpreadSketch reports x as a superspreader with a probability at most δ.*

Proof. A source x is reported as a superspreader only if $\hat{S}(x) \ge \phi\mathcal{S}$ and x is stored in one of its hashed buckets. We first consider the probability $\Pr[\hat{S}(x) \ge \phi\mathcal{S}]$. From the proof of Theorem 2, we have $\hat{S}(x) \le (1+\sigma)(R_i + S(x))$ for each row i, where $1 \le i \le r$. Thus,

$$\begin{aligned}
\Pr[\hat{S}(x) \ge \phi\mathcal{S}] &\le \Pr[(1+\sigma)(R_i + \tfrac{\epsilon\mathcal{S}}{1+\sigma}) \ge \phi\mathcal{S}, \forall_i] \\
&= \Pr[R_i \ge \tfrac{\phi-\epsilon}{1+\sigma}\mathcal{S}, \forall_i] \\
&\le \Pr[R_i \ge \epsilon\mathcal{S}, \forall_i] \text{ (due to } \epsilon \le \tfrac{\phi}{4} \text{ and } \sigma < 1) \\
&\le \tfrac{1}{2^r} = \delta \text{ (by Inequality (1)).}
\end{aligned}$$

We next consider the probability that x is stored in one of its hashed buckets. By Lemma 1, this probability is less than one. Combining both cases, the theorem follows. □

TABLE I
COMPARISON OF SPREADSKETCH WITH STATE-OF-THE-ART SKETCHES.

Sketches	r	w	Memory space	Per-packet update time	Detection time
DCS	$\log \frac{1}{\delta}$	$\frac{H}{\epsilon^2} \log((1+\log n)/\delta)$	$O(\frac{H}{\epsilon^2} \log^2 n \log((2+\log n)/\delta))$	$O(\log n \log \frac{1}{\delta})$	$O(\frac{H}{\epsilon^2} \log^2 n \log((2+\log n)/\delta))$
CDS	$\log \frac{1}{\delta}$	$\frac{2}{\epsilon}$	$O(\frac{m}{\epsilon} \log \frac{1}{\delta})$	$O(\log \frac{1}{\delta})$	$O(H^{\log(1/\delta)})$
VBF	$\log \log n$	$n^{1/\log \log n}$	$O(m(\log \log n) n^{1/\log \log n})$	$O(\log \log n)$	$O(H^{\log \log n})$
CMH	$\log \frac{1}{\delta}$	$\frac{2}{\epsilon}$	$O(\frac{m}{\epsilon} \log \frac{1}{\delta} + H \log n)$	$O(\log \frac{H}{\delta})$	$O(H)$
REV	$O(\frac{\log n}{\log \log n})$	$(\log n)^{\Theta(1)}$	$O(\frac{m(\log n)^{\Theta(1)}}{\log \log n})$	$O(\log n)$	$O(Hn \frac{3}{\log \log n} \log \log n)$
FAST	$4H \log \frac{4}{\delta}$	$1+\log \frac{n}{4H \log(4/\delta)}$	$O(Hm \log \frac{1}{\delta} \log \frac{n}{H \log(1/\delta)})$	$O(\log \frac{1}{\delta} \log \frac{n}{H \log(1/\delta)})$	$O(H \log^3 \frac{1}{\delta} \log(\frac{n}{H \log(1/\delta)}))$
SpreadSketch	$\log \frac{1}{\delta}$	$\frac{2}{\epsilon}$	$O(\frac{m+\log n+\log \log n}{\epsilon} \log \frac{1}{\delta})$	$O(\log \frac{1}{\delta})$	$O(H \log \frac{1}{\delta})$

D. Analysis for Network-Wide Superspreader Detection

We briefly discuss the memory space, performance, and accuracy of network-wide superspreader detection. Suppose that we deploy q measurement points, each of which runs a SpreadSketch instance with the same configuration parameters as in a single-sketch case. Since there are q SpreadSketch instances, the memory space is $O(\frac{q(m+\log n+\log \log n)}{\epsilon} \log \frac{1}{\delta})$ (i.e., q times the single-sketch case). The per-packet update time at each measurement point remains $O(\log \frac{1}{\delta})$. To recover all superspreaders, the controller takes $O(qrw) = O(\frac{q}{\epsilon} \log \frac{1}{\delta})$ time to merge q sketches and $O(rw) = O(\frac{1}{\epsilon} \log^2 \frac{1}{\delta})$ time to traverse all the buckets of the merged sketch, so the total detection time for returning all superspreaders is $O(\frac{1}{\epsilon} \log \frac{1}{\delta}(q + \log \frac{1}{\delta}))$. Finally, our network-wide detection operates on a $r \times w$ merged sketch, so its false negative and false positive rates follow Theorems 3 and 4 as in the single-sketch case, respectively.

E. Comparison with Existing Approaches

We compare SpreadSketch with several state-of-the-art sketches on superspreader detection (Section II-C), including Distinct-Count Sketch (DCS) [14], Connection Degree Sketch (CDS) [36], Vector Bloom Filter (VBF) [24], Count-Min-Heap (CMH) [7], RevSketch (REV) with distinct counting [31], [42], and Fast Sketch (FAST) [25], [26] with distinct counting. Table I shows the space and time complexities of all sketches in terms of ϵ, δ, n, m, and H (the maximum number of superspreaders that appear in an epoch). We assume that the distinct counters and bit arrays used in the sketches all have $O(1)$ time complexities. For CDS [36], the original paper does not discuss the table configuration with respect to accuracy parameters, so we set the numbers of rows and buckets of CDS as in SpreadSketch.

Space. DCS has high memory space as it includes the term $\frac{\log^2 n}{\epsilon^2}$. CMH, REV, FAST, and SpreadSketch all contain a $\log n$ term. However, the term refers to $\log n$ bits in CMH and SpreadSketch, while it refers to $\log n$ distinct counters in FAST and REV. It is not obvious whether SpreadSketch has smaller memory space than CDS and VBF. However, our evaluation (Section V) shows that SpreadSketch achieves higher accuracy than both CDS and VBF under the same memory space.

Per-packet update time. CMH incurs $\log \frac{1}{\delta}$ memory accesses to update the sketch and takes $O(\log H)$ time to access its heap if the source is a superspreader. DCS, REV, and FAST all have high update time complexities, which are proportional to $\log n$ (i.e., the key length). VBF extracts consecutive bits of each source key (in $O(1)$ time) to locate $O(\log \log n)$ hashed buckets. Both CDS and SpreadSketch have the same low per-packet update time.

Detection time. DCS, CDS, VBF, REV, and FAST all have high detection time complexities; in particular, the detection times of both CDS and VBF increase exponentially with the number of rows. CMH takes only $O(H)$ time to return all superspreaders and their estimated fan-outs from its heap. SpreadSketch takes $O(\log \frac{1}{\delta})$ time to estimate the fan-out of each superspreader, and hence $O(H \log \frac{1}{\delta})$ time in total.

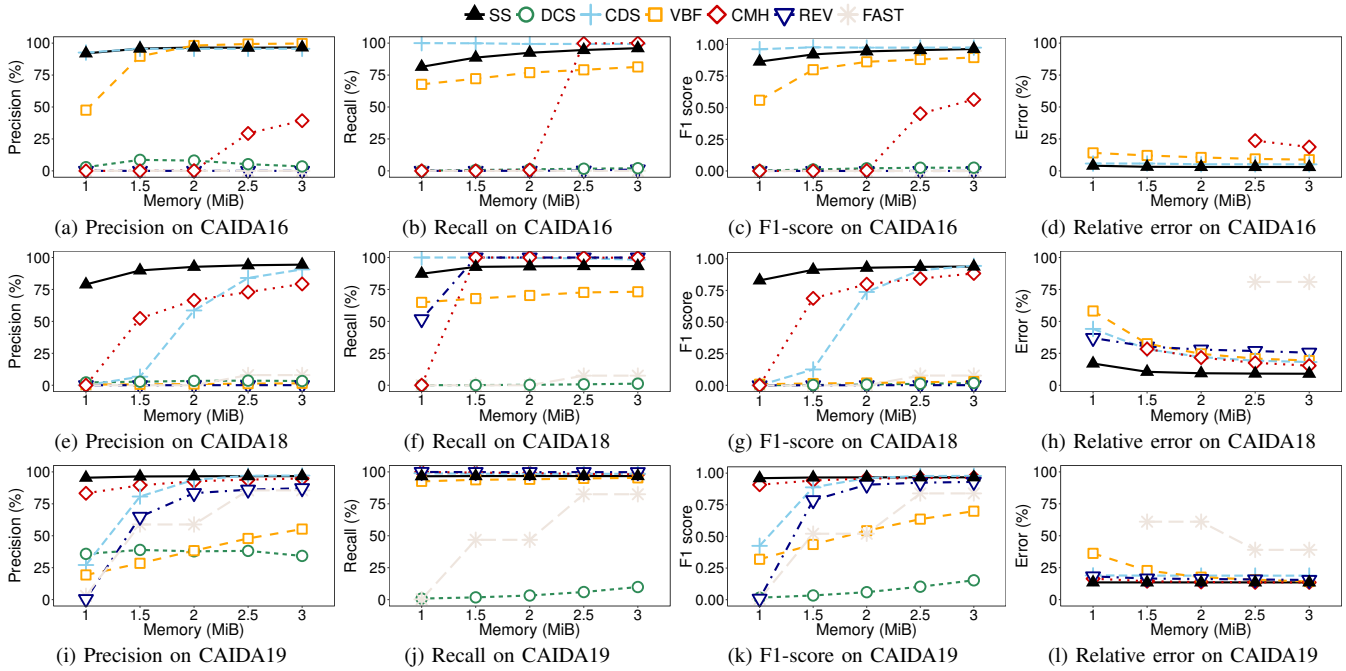
V. EVALUATION

We conduct trace-driven evaluation on real-world Internet traces and compare SpreadSketch with state-of-the-art sketches. We show that SpreadSketch achieves (i) high detection accuracy, (ii) high update and detection performance, and (iii) accurate network-wide superspreader detection. We further implement SpreadSketch in P4 [3] and present its microbenchmark performance on a Barefoot Tofino switch [1].

A. Setup

Traces. We consider three one-hour real-world traces, namely *CAIDA16*, *CAIDA18*, and *CAIDA19*, captured by CAIDA [2] on 10 GigE backbone links in years 2016, 2018, and 2019, respectively. We divide each trace into 60 one-minute epochs. We focus on the source-destination address pairs of IPv4 traffic only. The three traces have highly different statistical properties as well as skewness (Figure 1 in Section III-A): CAIDA16 (least skewed), CAIDA18 (moderately skewed), and CAIDA19 (most skewed) contain 0.46K, 1.31K, and 0.35K unique sources, as well as 0.74K, 5.28K, and 2.58K distinct pairs per epoch on average, respectively. We evaluate superspreader detection in each epoch and obtain averaged results over all epochs.

Parameter configurations. We compare SpreadSketch with the state-of-the-art sketches listed in Table I. For fair comparisons, we use the multiresolution bitmap [9] as the distinct counter in CMH, REV, and FAST. We fix a multiresolution bitmap as 438 bits, so that it can count up to 10,000 distinct items with $\sigma = 0.1$ (Section III-D). Also, we configure the same memory usage for all sketches. For SpreadSketch, we fix $r = 4$ rows and vary the number of buckets per row (i.e., w) for each given memory size. For other sketches, we tune r and w under the given memory size and choose the setting that maximizes the accuracy (F1-score). We tune the threshold for each trace to keep the number of true superspreaders in each



(a) Precision on CAIDA16 (b) Recall on CAIDA16 (c) F1-score on CAIDA16 (d) Relative error on CAIDA16

(e) Precision on CAIDA18 (f) Recall on CAIDA18 (g) F1-score on CAIDA18 (h) Relative error on CAIDA18

(i) Precision on CAIDA19 (j) Recall on CAIDA19 (k) F1-score on CAIDA19 (l) Relative error on CAIDA19

Fig. 4. (Experiment 1) Accuracy. We do not plot the relative errors for some settings if the corresponding recall is zero.

epoch as 100. In particular, we fix the heap size of CMH as 256 source keys, so as to provide sufficient space for storing candidate superspreaders.

Metrics. We consider the following metrics.

- *Precision*: the ratio of true superspreaders detected over all superpspreaders reported;
- *Recall*: the ratio of true superspreaders detected over all true superspreaders reported;
- *F1-score*: the harmonic average of precision and recall;
- *Relative error*: $\frac{1}{|D|}\sum_{x\in D}\frac{|\hat{S}(x)-S(x)|}{S(x)}$, where D is the set of true superspreaders detected;
- *Throughput*: the number of packets processed per second;
- *Detection time*: the time spent on recovering all superspreaders.

B. Results

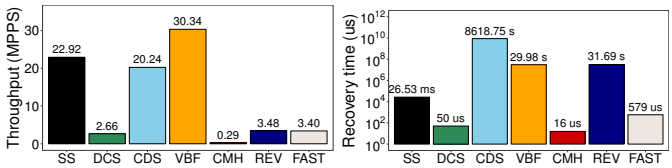
(Experiment 1) Accuracy. Figure 4 compares the accuracy of SpreadSketch with that of other sketches on all three traces versus the memory size (varied from 1 MiB to 3 MiB). We make several observations. First, CDS has the highest F1-score on CAIDA16, yet its precision drops greatly on CAIDA18 and CAIDA19 when the memory is no more than 1.5 MiB (e.g., near zero in Figure 4(e)). The reason is that both CAIDA18 and CAIDA19 have much more distinct pairs than CAIDA16, and CDS needs more buckets to distinguish the sources with large fan-outs. With insufficient buckets, CDS returns many false positives. Similar observations apply to VBF, which has a precision of near zero on CAIDA18 and below 0.56 on CAIDA19. Second, CMH, FAST, and REV all have a higher F1-score for more skewed traces (e.g., the lowest F1-score on CAIDA16, and the highest F1-score on CAIDA19). The reason is that with a higher skewness of fan-outs, they can distinguish more readily superspreaders from normal sources. Third, DCS has a nearly zero F1-score on all traces, as it requires more memory to report all superspreaders.

SpreadSketch achieves the highest F1-score in most cases. Its F1-score is 0.86-0.96, 0.82-0.93, and 0.96-0.97 on CAIDA16, CAIDA18, and CAIDA19, respectively; it is the only sketch that achieves an F1-score of above 0.9 when the memory size is at least 1.5 MiB. Although it has a lower F1-score than CDS on CAIDA16, SpreadSketch is generally much more robust than CDS and the other sketches on accuracy on all traces. Also, SpreadSketch achieves the lowest relative errors among all sketches on all traces.

(Experiment 2) Performance. We benchmark the performance of all sketches on a server equipped with an eight-core Intel Xeon E5-1630 3.70 GHz CPU and 16 GiB RAM. The server runs Ubuntu 14.04.5. Before running each experiment on a trace, we load the whole trace into memory to exclude any disk I/O overhead. We present only the results on CAIDA16, while the same observations are made on other traces. We fix the memory size of all sketches as 1 MiB. Our plots omit the error bars as the variances across epochs are negligible.

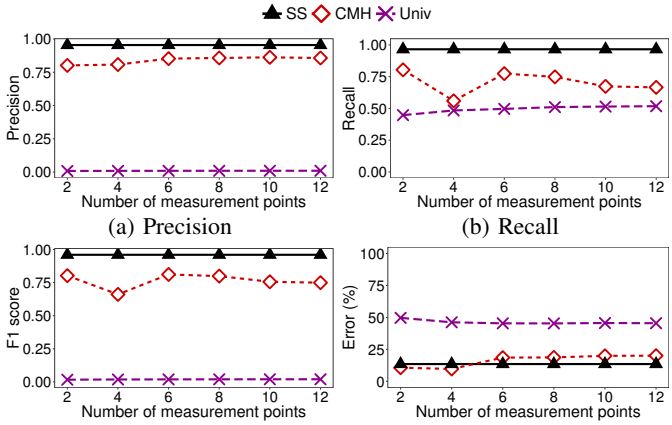
Figure 5(a) shows the update throughput of adding the source-destination pairs of a packet stream into a sketch (in million packets per second (MPPS)). SpreadSketch, CDS, and VBF all achieve a throughput of above 14.88 MPPS, implying that they can match a 10 Gb/s line-rate in software. VBF has the highest throughput as it uses the consecutive bits of a source key to locate buckets, while other sketches including SpreadSketch perform multiple hash computations to map a source to buckets. CMH has the lowest throughput, as it spends non-negligible time to estimate fan-outs and traverse the heap structure.

Figure 5(b) shows the detection time of returning all



(a) Update throughput (b) Detection time

Fig. 5. (Experiment 2) Performance.



(a) Precision (b) Recall

(a) F1-score (b) Relative error

Fig. 6. (Experiment 3) Network-wide detection.

superspreaders. SpreadSketch, DCS, CMH and FAST all return superspreaders in few milliseconds. CMH has the smallest detection time as it outputs superspreaders from its heap structure directly. In contrast, VBF and REV take around 30 s to recover superspreaders from their data structures, and CDS even takes over two hours. Overall, SpreadSketch achieves both high update and detection performance.

(Experiment 3) Network-wide detection. We now study network-wide detection, and also include the sketch-based network-wide measurement system UnivMon [27] in comparisons. For UnivMon, we replace all integer counters with multiresolution bitmaps for superspreader detection. We simulate a network-wide scenario (Section III-F) by partitioning the packets in an epoch of a trace to a given number of measurement points (i.e., the same source may appear in multiple measurement points). We present only the results on CAIDA19, while the results are similar for other traces. Also, among state-of-the-art sketches, we show only the results for CMH; for others, the accuracy remains identical as in single-point detection. We again fix the memory space of each sketch at each measurement point as 1 MiB.

Figure 6 shows the accuracy versus the number of measurement points. The accuracy of SpreadSketch is maintained regardless of the number of measurement points. In contrast, the F1-score of CMH varies with the number of measurement points and generally shows a downtrend. The reason is that CMH only keeps (in its heap structure) the source keys whose fan-outs exceed a pre-specified threshold at each measurement point, but it may likely miss the superpspreaders that show small fan-outs at most measurement points. UnivMon achieves almost a zero F1-score in all cases, as it maintains information for different traffic statistics and hence requires much more memory to achieve high accuracy.

TABLE II
SWITCH RESOURCE USAGE OF SPREADSKETCH (PERCENTAGES IN BRACKETS ARE FRACTIONS OF TOTAL RESOURCE USAGE).

SRAM (KiB)	No. stages	No. actions	No. ALUs	PHV size (bytes)
256 (1.67%)	6 (50%)	20 (nil)	6 (12.5%)	108 (14%)

(Experiment 4) SpreadSketch in hardware. We implement SpreadSketch in P4 [3] (with less than 500 lines of code) and compile it in the Barefoot Tofino chipset [1]. Our implementation realizes each row of SpreadSketch as an array of registers that can be directly updated in the switch data plane via stateful ALUs. We generate the hash string of each source-destination pair in a dedicated match-action table. We then count the number of leading zeros of the hash string using a longest-prefix-match table. If a hash string matches one entry of the table, the action of that entry will return the corresponding level value. To fit SpreadSketch in limited switch memory, we set $r = 3$, $w = 2048$, and $m = 128$. We find that SpreadSketch achieves an F1-score of over 0.9 for an epoch length of one second on all CAIDA traces.

Table II shows the switch resource usage of SpreadSketch in SRAM consumption, the numbers of physical stages, actions, and stateful ALUs (all of which measure computational resources), as well as the packet header vector (PHV) size (which measures the message size across stages). SpreadSketch uses 256 KiB of SRAM, which accounts for only 1.67% of the total SRAM. We can place all the tables, registers, and ALU operations for managing SpreadSketch in the data plane in six physical stages (half of the total stages of the Tofino chipset). However, SpreadSketch still leaves sufficient resources in each occupied stage for other applications since its overall consumptions of SRAM and ALUs are limited. Our prototype contains 20 actions in total to process packets, including hash computations and the updates of register arrays. To perform transactional read-test-write operations on multiple buckets for each source-destination pair, SpreadSketch consumes only six (12.5% of total) stateful ALUs. The PHV size in our prototype is 108 bytes (14% of total PHV resources), nearly half of which are needed to store packet header information for packet forwarding. We also validate that SpreadSketch can process packets at line-rate on a Tofino switch.

VI. CONCLUSIONS

This paper designs a new invertible sketch data structure called SpreadSketch for network-wide superspreader detection. We show via theoretical analysis and trace-driven evaluation that SpreadSketch achieves high memory efficiency, high update and detection performance, as well as high detection accuracy. We further implement SpreadSketch in P4 and demonstrate its feasible deployment in commodity hardware switches.

Acknowledgments: The work was supported by Research Grants Council of Hong Kong (GRF 14204017), National Key R&D Program of China (2019YFB1802600), National Natural Science Foundation of China (61802365), and CAS Pioneer Hundred Talents Program. The corresponding author is Qun Huang.

REFERENCES

[1] Barefoot's Tofino. https://barefootnetworks.com/products/brief-tofino/.

[2] CAIDA. http://www.caida.org/data/passive/trace_stats/.

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

[4] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner. Heavy Hitters in Streams and Sliding Windows. In *Proc. of IEEE INFOCOM*, 2016.

[5] J. Cao, Y. Jin, A. Chen, T. Bu, and Z.-L. Zhang. Identifying High Cardinality Internet Hosts. In *Proc. of IEEE INFOCOM*, 2009.

[6] 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.

[7] G. Cormode and S. Muthukrishnan. Space Efficient Mining of Multigraph Streams. In *Proc. of ACM PODS*, 2005.

[8] Z. Durumeric, M. Bailey, and J. A. Halderman. An Internet-Wide View of Internet-Wide Scanning. In *Proc. of USENIX Security Symposium*, 2014.

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

[10] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High Speed Links. Technical report, UCSD technical report CS2003-0738, 2003.

[11] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and Elastic DDoS Defense. In *Proc. of USENIX Security Symposium*, 2015.

[12] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The Analysis of a Near-optimal Cardinality Estimation Algorithm. In *Analysis of Algorithms*, 2007.

[13] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[14] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani. Streaming Algorithms for Robust, Real-Time Detection of DDoS Attacks. In *Proc. of IEEE ICDCS*, 2007.

[15] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L. Uden, and X. Li. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. In *Proc. of USENIX ATC*, pages 909–921, 2018.

[16] H. Huang, Y.-E. Sun, S. Chen, S. Tang, K. Han, J. Yuan, and W. Yang. You Can Drop but You Can't Hide: K-persistent Spread Estimation in High-speed Networks. In *Proc. of IEEE INFOCOM*, 2018.

[17] 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.

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

[19] N. Kamiyama, T. Mori, and R. Kawahara. Simple and Adaptive Identification of Superspreaders by Flow Sampling. In *Proc. of IEEE INFOCOM*, 2007.

[20] D. M. Kane, J. Nelson, and D. P. Woodruff. An Optimal Algorithm for the Distinct Elements Problem. In *Proc. of ACM PODS*, 2010.

[21] D. E. Knuth. *The Art of Computer Programming, Volume 4*. Addison-Wesley Professional, 2015.

[22] T. Li, S. Chen, W. Luo, M. Zhang, and Y. Qiao. Spreader Classification Based on Optimal Dynamic Bit Sharing. *IEEE/ACM Trans. on Networking*, 21(3):817–830, 2013.

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

[24] W. Liu, W. Qu, J. Gong, and K. Li. Detection of Superpoints Using a Vector Bloom Filter. *IEEE Trans. on Information Forensics and Security*, 11(3):514–527, 2016.

[25] Y. Liu, W. Chen, and Y. Guan. A Fast Sketch for Aggregate Queries over High-Speed Network Traffic. In *Proc. of IEEE INFOCOM*, 2012.

[26] Y. Liu, W. Chen, and Y. Guan. Identifying High-Cardinality Hosts from Network-Wide Traffic Measurements. *IEEE Trans. on Dependable and Secure Computing*, 13(5):547–558, 2016.

[27] 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.

[28] R. Martin. Snort: Lightweight Intrusion Detection for Networks. In *Proc. of USENIX LISA*, 1999.

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

[30] D. Plonka. FlowScan: A Network Traffic Flow Reporting and Visualization Tool. In *Proc. of USENIX LISA*, 2000.

[31] 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.

[32] S. Sen and J. Wang. Analyzing Peer-to-Peer Traffic Across Large Networks. In *Proc. of ACM SIGCOMM*, 2002.

[33] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proc. of USENIX OSDI*, 2004.

[34] L. Tang, Q. Huang, and P. P. Lee. MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams. In *Proc. of IEEE INFOCOM*, 2019.

[35] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *Proc. of NDSS*, 2005.

[36] P. Wang, X. Guan, T. Qin, and Q. Huang. A data streaming method for monitoring host connection degrees of high-speed links. *IEEE Trans. on Information Forensics and Security*, 6(3):1086–1098, 2011.

[37] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications . *ACM Trans. on Database Systems*, 15(2):208–229, 1990.

[38] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen. Estimating the Persistent Spreads in High-Speed Networks. In *Proc. of IEEE ICNP*, 2014.

[39] 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.

[40] M. Yoon and S. Chen. Detecting Stealthy Spreaders by Random Aging Streaming Filters. *IEICE Trans. on communications*, 94(8):2274–2281, 2011.

[41] M. Yoon, T. Li, S. Chen, and J.-K. Peir. Fit a Compact Spread Estimator in Small High-Speed Memory. *IEEE/ACM Trans. on Networking*, 19(5):1253–1264, 2011.

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

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

[44] Y. Zhou, Y. Zhou, M. Chen, and S. Chen. Persistent Spread Measurement for Big Network Data Based on Register Intersection. *Proc. of ACM on Measurement and Analysis of Computing Systems*, 1(1):15, 2017.