A High-Performance Invertible Sketch for Network-Wide Superspreader Detection

Lu Tang, Yao Xiao, Qun Huang, Patrick P. C. Lee

***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 readily 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. We further extend SpreadSketch with a fast and small data structure that filters out the packets of high-frequency connections from sketch processing, so as to improve the update performance of SpreadSketch while maintaining the accuracy guarantees. Trace-driven evaluation shows that SpreadSketch achieves higher accuracy and performance over state-of-the-art sketches and remains accurate in detecting real-world worms and DDoS attacks. 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 [47] and detection of malicious attacks (e.g., DDoS attacks [19], port scanning [15], and worm propagation [48]). For example, superspreaders may refer to the infected hosts that connect to many other hosts for worm propagation [48], or the servers overwhelmed by a botnet of zombie hosts under DDoS attacks [19]. Despite many efforts in superspreader detection over decades, superspreaders (e.g., DDoS attacks) remain widespread in modern production networks [19].

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 [41]. A simple network-wide detection approach is to maintain complete per-flow states [42], [44], 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., [14], [23], [38], [40], [53], [60]) 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 [21]). 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

The work was supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (62172007), the Fundamental Research Funds for the Central Universities (20720210072), the Natural Science Foundation of Fujian Province of China (2021J05002), and Research Grants Council of Hong Kong (GRF 14204017).

An earlier version of this paper appeared at [51]. In this extended version, we extend SpreadSketch with a small data structure, namely the HP-Filter, to enhance its update performance. We also add new evaluation results on the effectiveness of SpreadSketch in real-world attack detection.

L. Tang and Y. Xiao are with the Department of Computer Science and Technology, Xiamen University, Xiamen, China (Emails: tanglu@xmu.edu.cn, yaoxiao@stu.xmu.edu.cn).

Q. Huang is with the Department of Computer Science and Technology, Peking University, Beijing, China (Email: huangqun@pku.edu.cn).

P. P. C. Lee is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China (Email:pclee@cse.cuhk.edu.hk).

Corresponding author: Patrick P. C. Lee.

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 [27], [28], [37], [41], [43], [57], [60]. 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 (Section III).
- 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 (Section IV).
- We extend SpreadSketch with a fast and small data structure, called the HP-Filter, that filters out the packets of high-frequency connections without processing the packets in the sketch data structure. The HP-Filter improves the update performance and maintains the accuracy of SpreadSketch, with a trade-off of slightly increasing the memory usage. We also present theoretical analysis on the extended design of SpreadSketch with the HP-Filter (Section V).
- 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. Adding the HP-Filter to SpreadSketch further increases the update throughput by up to 54.5%. SpreadSketch also achieves high accuracy in detecting real-world worms and DDoS attacks (Section VI).
- We prototype SpreadSketch in P4 [4] and compile it in the Barefoot Tofino chipset [1]. We present microbenchmark results on SpreadSketch to justify its feasible deployment in commodity hardware switches (Section VI).

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

II. Background and Related Work

We formulate the superspreader detection problem and review the related work. Table I summarizes the major notation in this paper.

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., [17], [20], [21], [55]) 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.

TABLE I
Major notation used in the paper.

Notation	Description
Defined in Section II	
(x,y)	a source-destination pair
$[n]$	domain of a key
ϕ	fraction threshold, where $0 < \phi < 1$
$S(x)$	fan-out of a source key x (e.g., x represents a source IP)
$\mathcal{S}$	total fan-out of all sources in an epoch
Defined in Section III	
r	number of rows in a sketch
w	number of buckets in each row in a sketch
$B(i,j)$	bucket at i-th row and j-th column ($1 \leq i \leq r$ and $1 \leq j \leq w$)
$V_{i,j}$	sum of fan-out of all sources hashed to $B(i,j)$
$k_{i,j}$	candidate superspreader in $B(i,j)$
$L_{i,j}$	maximum level in $B(i,j)$
h_i	hash function of the i-th row
h^*	globle hash function
c	number of bitmaps in multiresolution bitmap
b	number of bits in the first $c-1$ bitmaps of multiresolution bitmap
b'	number of bits in the last bitmap of multiresolution bitmap
q	number of measurement points
Defined in Section IV	
ε	approximation parameter of sketch, where $0 < \varepsilon \leq \frac{\phi}{4} < 1$
δ	error probability of sketch, where $0 < \delta < 1$
σ	error factor of distinct counter
m	number of bits in each distinct counter
H	maximum number of superspreaders in an epoch
Defined in Section IV	
t	number of units in HP-Filter
$U(i)$	the i-th unit in HP-Filter ($1 \leq i \leq t$)
P_i	heavy repeating pair in $U(i)$
I_i	indicator counter in $U(i)$

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 [41].
- **Small and static memory usage:** The sketch incurs small memory footprints, which are essential for the deployment in both software and hardware [28]. Also, its memory resources can be statically allocated in advance to avoid dynamic memory management overhead [50].
- **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.
- **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.

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 [42] and FlowScan [44]) maintain all active connections for each source to identify any port scans or DDoS attacks. To improve memory efficiency, Estan et al. [17] 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* [11], [32], [52] 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 [37]. Also, some approaches [11], [32] maintain sampled flows in chained hash tables, which have high memory access overhead over the linked lists of buckets.

Streaming. Zhao et al. [62] combine sampling (which filters hosts with small flow counts) and streaming (which estimates the fan-out of each sampled source). Some approaches [30], [36], [49], [59] estimate the fan-outs of sources in tight memory space by sharing counter bits among sources. The above solutions design compact data structures 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 [23] extends the idea of Probabilistic Counting [21] 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 [53] reconstructs host addresses associated with large fan-outs based on the Chinese Remainder Theorem. Vector Bloom Filter [38] 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 [14] extends Count-Min sketch [13] with an external heap for tracking superspreaders and associates each bucket with a distinct counter (e.g., [17], [20], [21], [55]). OpenSketch [60] combines Reversible Sketch [46] with bitmap algorithms [17], while Liu et al. [40] combine Fast Sketch [39] with the optimal distinct counter [33], for superspreader detection. However, existing invertible frequency-based sketches generally have high processing overhead [50]: Count-Min-Heap not only incurs high memory access overhead for heap updates, but also needs to estimate the fan-out of the source key for each packet update in order to determine if the source key should be kept in the heap; 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., [6], [24], [50]) 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 [27], [28], [37], [41], [43], [57], [60] propose unified frameworks for general measurement tasks, and some of them [27], [41], [43], [60] 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. [58] design a random aging streaming filter to find stealthy spreaders that send low-rate malicious packets. Xiao et al. [56] and Zhou et al. [64] study the estimation of persistent fan-outs over a number of epochs. Huang et al. [26] further address the

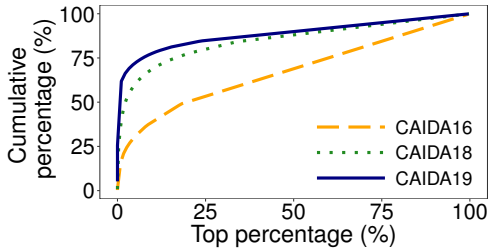


Fig. 1. Cumulative fan-out ratios of the top-percentages of sources (i.e., the sum of fan-outs of the top-percentage of sources over the total fan-outs of all sources) for three real-world Internet traces.

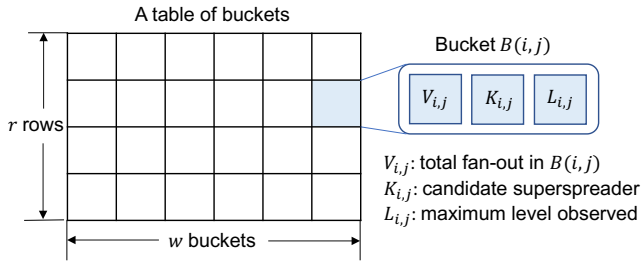


Fig. 2. Data structure of SpreadSketch.

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.

Hardware solutions. Recent studies [12], [35], [54], [63] customize general measurement solutions, including super-spreader detection, for programmable switches. Elastic Trie [35] iteratively tracks the IP prefixes in the data plane and adopts a push-based approach to inform the controller about network events. Martini [54] exports primitives for measurement tasks and performs measurement and control decisions entirely in the data plane to reduce the control-loop time. Newton [63] allows operators to deploy measurement tasks on demand by installing match-action rules dynamically. BeauCoup [12] supports multiple heterogeneous distinct counting queries directly in the data plane. In contrast, SpreadSketch's design targets the deployment in both software and hardware.

III. SpreadSketch Design

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

A. Main Idea

SpreadSketch is a non-trivial extension of the classical *Count-Min Sketch* [13]. 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., [17], [20], [21], [55]). 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 often highly skewed [26], [29], [31], 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 packet traces collected from the Internet (see Section VI 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. In addition to the Internet traces, we also observe skewness in the traces from university data centers [22] and enterprise networks [45] (see the digital supplementary file for details).

Accurate fan-out estimation via hash strings. Inspired by Probabilistic Counting [21], 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 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

1: **procedure** UPDATE(x,y)
2: $l \leftarrow$ 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** $L_{i,h_i(x)} \leq l$ **then**
6: $(K_{i,h_i(x)}, L_{i,h_i(x)}) \leftarrow (x,l)$
7: **end if**
8: **end for**
9: **end procedure**
10: **procedure** QUERY(x)
11: **return** $\hat{S}(x) \leftarrow \min_{1\leq i\leq r}\{|V_{i,h_i(x)}|\}$
12: **end procedure**
13: **procedure** COUNT(V,x,y)
14: $l \leftarrow$ length of most significant 0-bits of $h^*(x,y)$
15: **if** $l < c-1$ **then**
16: $p \leftarrow h^*(x,y) \mod b$
17: $V[l][p] \leftarrow 1$
18: **else**
19: $p \leftarrow h^*(x,y) \mod b'$
20: $V[c-1][p] \leftarrow 1$
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: $V_{i,j} \leftarrow V^1_{i,j} \cup V^2_{i,j} \cdots \cup V^q_{i,j}$
27: $K_{i,j} \leftarrow K^{k^*}_{i,j}$, where $k^* = \arg\max_{1\leq k\leq q}\{L^k_{i,j}\}$
28: $L_{i,j} = \max_{1\leq k\leq q}\{L^k_{i,j}\}$
29: **end for**
30: **end for**
31: **end procedure**

Fig. 3. Main operations of SpreadSketch.

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 ($1 \leq i \leq r$) 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 [20], [34].

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 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 ($1 \leq i \leq r$), 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 $L_{i,h_i(x)} \leq l$, 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 $1 \leq i \leq r$ (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* [17], which supports multiset operations for network-wide detection (Section III-F) and can be readily implemented in hardware (Section VI). Compared with other distinct counters such as Linear Counting [55], HyperLogLog [20], and K-Minimum Values [8], the multiresolution bitmap maintains more stable accuracy for various counting ranges using a small number of bits (see the digital supplementary file). Recent work shows that short approximate counters can be combined with sketches for faster and more space-efficient measurement [7]; we pose the study of such a combination issue as future work.

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 [17].

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 [18]. 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 [18] 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 [17], 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 $K^k_{i,j}$'s; 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 (worst-case) analysis on SpreadSketch in memory space, performance, and accuracy. We configure SpreadSketch with three parameters: ε, δ, and σ ($0 < \varepsilon, \delta, \sigma < 1$), 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 $w = \frac{2}{\varepsilon}$, where the logarithm base is 2. Our analysis also assumes $\varepsilon \le \frac{\phi}{4}$ to provide provable error bounds. Given σ, we can derive the minimum memory space m (in bits) for a distinct counter (Section III-D).

In the interest of space, we refer readers to the digital supplementary file for the detailed proofs.

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

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 [17] that we use and other distinct counters [33], [55].

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* $1-\delta$, $\hat{S}(x) \le (1+\sigma)(S(x)+\varepsilon\mathcal{S})$.

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* $1-\delta$.

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* $1-\delta$.

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

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 $O(\frac{q(m+\log n+\log\log n)}{\varepsilon}\log\frac{1}{\delta})$ (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 $O(qrw) = O(\frac{q}{\varepsilon}\log\frac{1}{\delta})$ time to merge q sketches and $O(rw) = O(\frac{1}{\varepsilon}\log^2\frac{1}{\delta})$ time to traverse all the buckets of the merged sketch, so the total detection time for returning all superspreaders is $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}(q+\log\frac{1}{\delta}))$. 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) [23], Connection Degree Sketch (CDS) [53], Vector Bloom Filter (VBF) [38], Count-Min-Heap (CMH) [14], RevSketch (REV) with distinct counting [46], [60], and Fast Sketch (FAST) [39], [40] with distinct counting. Table II 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

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

Sketches	r	w	Memory space	Per-packet update time	Detection time
DCS	$\log\frac{1}{\delta}$	$\frac{H}{\varepsilon^2}\log((1+\log n)/\delta)$	$O(\frac{H}{\varepsilon^2}\log^2 n\log((2+\log n)/\delta))$	$O(\log n\log\frac{1}{\delta})$	$O(\frac{H}{\varepsilon^2}\log^2 n\log((2+\log n)/\delta))$
CDS	$\log\frac{1}{\delta}$	$\frac{2}{\varepsilon}$	$O(\frac{m}{\varepsilon}\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$	$O(H^{\log(1/\delta)})$
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	$\log\frac{1}{\delta}$	$\frac{2}{\varepsilon}$	$O(\frac{m}{\varepsilon}\log\frac{1}{\delta}+H\log n)$	$O(\log\frac{H}{\delta})$	$O(H)$
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	$\log\frac{1}{\delta}$	$\frac{2}{\varepsilon}$	$O(\frac{m+\log n+\log\log n}{\varepsilon}\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$	$O(H\log\frac{1}{\delta})$

$O(1)$ time complexities. For CDS [53], 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 $\frac{\log^2 n}{\varepsilon^2}$. 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 VI) 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. UPDATE IMPROVEMENTS OF SPREADSKETCH

Reducing the per-packet update time in sketches is critical for superspreader detection, especially in high-speed networks. If the update speed cannot match up with the line-rate, there will be packet drops in on-path sketch processing. While SpreadSketch can achieve the 10 Gb/s line-rate (Section VI-B), its update time may become the performance bottleneck in networks with higher bandwidth (e.g., 100 Gb/s [25]). In this section, we propose an additional data structure to improve the update performance of SpreadSketch for high-speed networks. We also perform theoretical analysis on the update performance of the new SpreadSketch design. Note that since modern switch ASICs (e.g., Tofino [1]) already guarantee line-rate packet processing for the deployment of SpreadSketch in the data plane (see Experiment 7), our improved design is mainly for the deployment on CPU-based server platforms.

A. Heavy-Pair Filter (HP-Filter)

Design overview. Recall that SpreadSketch updates a packet stream of source-destination pairs into its sketch structure. Our insight is that we can further improve its update performance by filtering out the redundant pairs that appear more than once. Specifically, we refer to a pair as a *new pair* if it has never appeared in the stream before, or as a *repeating pair* otherwise. Repeating pairs do not contribute to the counting of the fan-out of a source, but are instead treated as noises. Currently, SpreadSketch differentiates the new and repeating pairs based on the distinct counter in each bucket. This causes SpreadSketch to spend non-negligible time to process repeating pairs. Thus, our idea is to filter out most repeating pairs at the input of SpreadSketch and pass the remaining pairs to SpreadSketch for further processing.

A straightforward way to filter out the repeating pairs is to use the Bloom filter [9], a space-efficient probabilistic bitmap structure that tests whether an element is present in a set. In the Bloom filter, each new pair is hashed into a subset of bits and sets the hashed bits; if all bits are already set, it implies that the pair is a repeating pair. However, we argue that Bloom filter is not well suited for superspreader detection in SpreadSketch, with two reasons. First, the Bloom filter has false positives due to hash collisions. It can falsely treat a new pair as a repeating pair, causing SpreadSketch to miss the new pair. Second, to minimize the false positive rate, the Bloom filter size should be pre-configured to be linear with the number of distinct pairs. This requires the knowledge of the number of distinct pairs in a stream a priori, which is infeasible in practice. For example, when a DDoS attack happens, a large number of distinct pairs emerge in a short period of time. If we do not address the attack scenario and the Bloom filter is configured with an underestimated size, it is easily saturated and causes SpreadSketch to miss many new pairs.

Our idea is to filter out repeating pairs from a heavy hitter detection perspective (Section II-D). Prior studies show that a small number of large flows (i.e., flows with significant amounts of traffic) dominate in IP traffic [16], [61], implying that a small number of source-destination pairs appear with high frequencies. We refer to the high-frequency pairs as the *heavy pairs*; note that a heavy pair is also a repeating pair since it must appear more than once in order for being treated as a heavy pair. If we can identify and filter the heavy pairs, we can eliminate a large fraction of repeating pairs from being updated in SpreadSketch. To this end, we propose a filter structure called the *HP-Filter*,

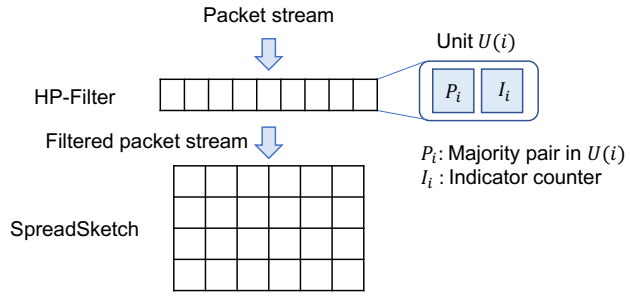


Fig. 4. Integration of the HP-Filter with SpreadSketch.

```
1:  procedure OPTUPDATE(x, y)
2:      if P_{g(x,y)} = (x, y) then
3:          I_{g(x,y)} ← I_{g(x,y)} + 1
4:          return
5:      else
6:          UPDATE(x, y)
7:          I_{g(x,y)} ← I_{g(x,y)} − 1
8:          if I_{g(x,y)} < 0 then
9:              I_{g(x,y)} ← 1
10:             P_{g(x,y)} ← (x, y)
11:         end if
12:     end if
13: end procedure
```

Fig. 5. The new update operation of HP-SpreadSketch.

which applies the *majority voting algorithm (MJRTY)* [10] to find the heavy pairs. MJRTY is a streaming method that is proven to always find the majority item whose frequency is more than half the total frequencies in a stream. It has been used in detecting heavy hitters [50], and we extend its idea to filter heavy pairs.

Algorithm details. Figure 4 shows the data structure of the HP-Filter and its integration with SpreadSketch. The HP-Filter comprises t units. Each unit $U(i)$ $(1 \le i \le t)$ contains two fields: (i) P_i, which stores the current heavy pair in $U(i)$; and (ii) I_i, which is an indicator counter for checking whether P_i should be updated via MJRTY. We hash each pair in a packet stream to one of the t units in the HP-Filter. Our idea is to apply MJRTY to track the heavy pair in each unit, assuming that the heavy pair dominates in the traffic among all pairs that are hashed to the same unit.

Figure 5 describes the update process of SpreadSketch coupled with the HP-Filter, which we refer to as *HP-SpreadSketch*. The idea is that if we find that a pair is determined to be a heavy pair according to the HP-Filter, we do not insert it into SpreadSketch. Initially, all units in the HP-Filter are set to zeros. For each pair (x,y), we hash it into one of the units $U(i)$ in the HP-Filter with some hash function g. We check if the pair should be filtered via MJRTY: if P_i equals (x,y), meaning that (x,y) is a candidate heavy pair in $U(i)$, we increment I_i by one and return (Lines 2-4); otherwise, we decrement I_i by one and insert (x,y) into SpreadSketch (Lines 5-7). If I_i is less than zero, meaning that the pair in P_i is no longer a heavy pair in $U(i)$ according to MJRTY, we reset I_i to one and set P_i to (x,y) (Lines 8-11).

B. Theoretical Analysis on HP-SpreadSketch

We present the theoretical analysis on HP-SpreadSketch. We show that HP-SpreadSketch maintains the accuracy of SpreadSketch (Theorem 5) and improves the throughput of SpreadSketch when the traffic is highly frequency-skewed (Theorem 6). We further show the complexities of memory space, update time, and detection time of HP-SpreadSketch (Theorem 7). The proofs are in the digital supplementary file.

Theorem 5. *HP-SpreadSketch has the same theoretical guarantee on accuracy as SpreadSketch.*

We study the per-packet update cost of HP-SpreadSketch under skewed traffic. Suppose that in an IP packet stream, the top-k heavy pairs account for at least a fraction p of the total number of packets N, where the parameters k and p determine the skewness of the packet stream. For example, for a fixed k, a larger p implies that the stream is more frequency-skewed. For the HP-Filter with t units, suppose that k' out of k heavy pairs ($k' < k$) have no hash collision with each other after the top-k heavy pairs are hashed to HP-Filter, while the k' heavy pairs account for a fraction p' of the total number of packets ($p' < p$). Also, suppose that the per-packet update cost in the HP-Filter is the same as the cost of updating a bucket in SpreadSketch. Let the per-packet update cost of the HP-Filter be one. Then the per-packet update cost of SpreadSketch is r.

Theorem 6 shows the condition when HP-SpreadSketch has a lower per-packet update cost than SpreadSketch. Our analysis assumes that each of the k' non-colliding heavy pairs is the majority pair in its hashed unit in the HP-Filter. Such an assumption can be justified for a sufficiently large number t of units in the HP-Filter, as the non-heavy pairs are spread across all the units and have limited contribution in the number of packets to each unit.

Theorem 6. *HP-SpreadSketch has a lower per-packet update cost than SpreadSketch when* $p' > \frac{2t[1-(1-1/t)^k]-k}{N} + \frac{\{2t[1-(1-1/t)^k]-k\}(1-p)}{t} + \frac{1}{r}$.

We briefly discuss the likelihood when the inequality in Theorem 6 holds. Suppose that the parameters r, t, N, and k are fixed. When p increases (i.e., the traffic becomes more frequency-skewed), the right-hand side of the inequality decreases while the left-hand side p' increases. Thus, the likelihood that the inequality holds increases with p.

Theorem 7 summarizes the worst-case complexity of HP-SpreadSketch in terms of memory space, update time, and detection time.

Theorem 7. *The memory space of HP-SpreadSketch is* $O(\frac{m+\log n+\log\log n}{\varepsilon}\log\frac{1}{\delta} + t\log n + t)$. *The per-packet update time is* $O(\log\frac{1}{\delta})$, *while the detection time of returning all superspreaders is* $O(\frac{1}{\varepsilon}\log^2\frac{1}{\delta})$.

HP-SpreadSketch significantly improves the update performance over SpreadSketch, with a trade-off of (slightly) increasing the memory usage due to the addition of the HP-Filter atop SpreadSketch. From our evaluation (Section VI-B), HP-SpreadSketch improves the update throughput up to at least 50%, while the HP-Filter itself accounts for less than 5% of the

total memory usage of HP-SpreadSketch. On the other hand, increasing the size of the HP-Filter can degrade the accuracy, as less memory will be allocated for SpreadSketch.

VI. 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 (Experiment 1), (ii) high update and detection performance (Experiment 2), and (iii) accurate network-wide superspreader detection (Experiment 3). We also demonstrate the update throughput gain of HP-SpreadSketch (Experiments 4-5) and the accurate detection of SpreadSketch in real-world attacks (Experiment 6). We further implement SpreadSketch in P4 [4] and present its microbenchmark performance on a Barefoot Tofino switch [1] (Experiment 7).

In the digital supplementary file, we report additional evaluation results on (i) using different distinct counters, including: Linear Counting [55], HyperLogLog [20], and K-Minimum Values [8], and the multiresolution bitmap [17]; (ii) using K-Minimum Values [8] for the distinct counters; and (iii) the comparisons with BeauCoup [12], a framework that supports multiple distinct counting queries (including superspreader detection) with constant memory accesses per packet.

A. Setup

Traces. We consider two sets of packet traces in our evaluation.

We consider three one-hour real-world IP packet traces, namely *CAIDA16*, *CAIDA18*, and *CAIDA19*, captured by CAIDA [2] on 10 GigE backbone links in the Internet 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.

We also consider two real-world attack traces to evaluate SpreadSketch in real-time attack detection. The first one is the *Witty Internet Worm* trace [5], which is monitored by the UCSD Network Telescope in year 2004 and contains the packets from the computers being infected by the Witty worm. The second one is the *CAIDA DDoS attack* trace [3], which is captured by CAIDA in year 2007 and contains packets from a DDoS attack. We divide each trace into one-minute epochs and evaluate superspreader detection in each epoch.

Parameter configurations. We compare SpreadSketch with the state-of-the-art sketches listed in Table II. For fair comparisons, we use the multiresolution bitmap [17] 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 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.

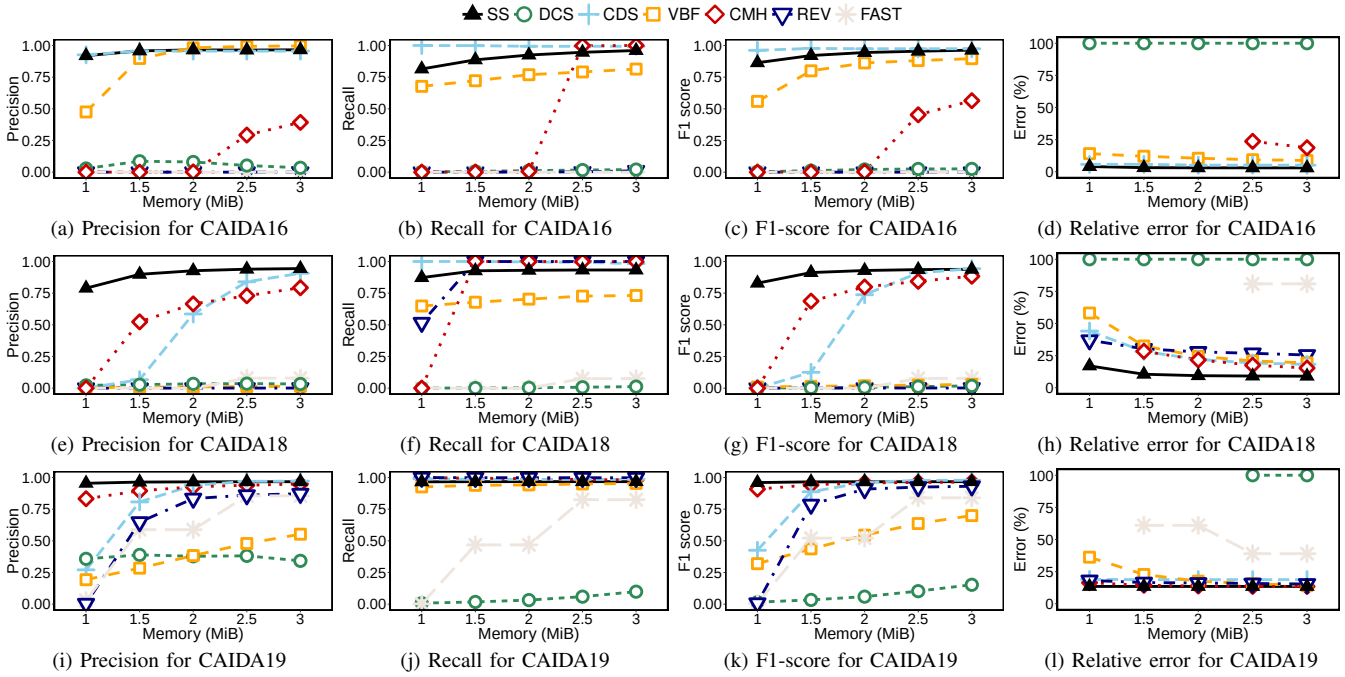
- *Precision*: the ratio of true superspreaders detected over all superspreaders reported;
- *Recall*: the ratio of true superspreaders detected over all true superspreaders reported;
- *F1-score*: the harmonic average of precision and recall;
- *Relative error*: $\frac{1}{|D|}\sum_{x\in D}\frac{|\hat{S}(x)-S(x)|}{S(x)}$,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

B. Results

(Experiment 1) Accuracy. Figure 6 compares the accuracy of SpreadSketch (SS) 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 for CAIDA18 and CAIDA19 when the memory is no more than 1.5 MiB (e.g., near zero in Figure 6(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 for CAIDA19. Second, CMH, FAST, and REV all have a higher F1-score for more skewed traces (e.g., the lowest F1-score for CAIDA16, and the highest F1-score for CAIDA19). The reason is that with 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 for CAIDA16, CAIDA18, and CAIDA19, respectively; it is the only sketch that achieves an F1-score above 0.9 when the memory size is at least 1.5 MiB. Although it has a lower F1-score than CDS for 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.

We further compare the accuracy of all sketches by varying the skewness in the fan-out of each source. Specifically, we extract each source IP in CAIDA19 and randomly generate its destination addresses to construct a set of synthetic traces. In each synthetic trace, the fan-out of each source IP follows a Zipf distribution with some skewness parameter, which we vary in our evaluation. As the skewness parameter increases (i.e., a more skewed distribution), there exist a smaller number

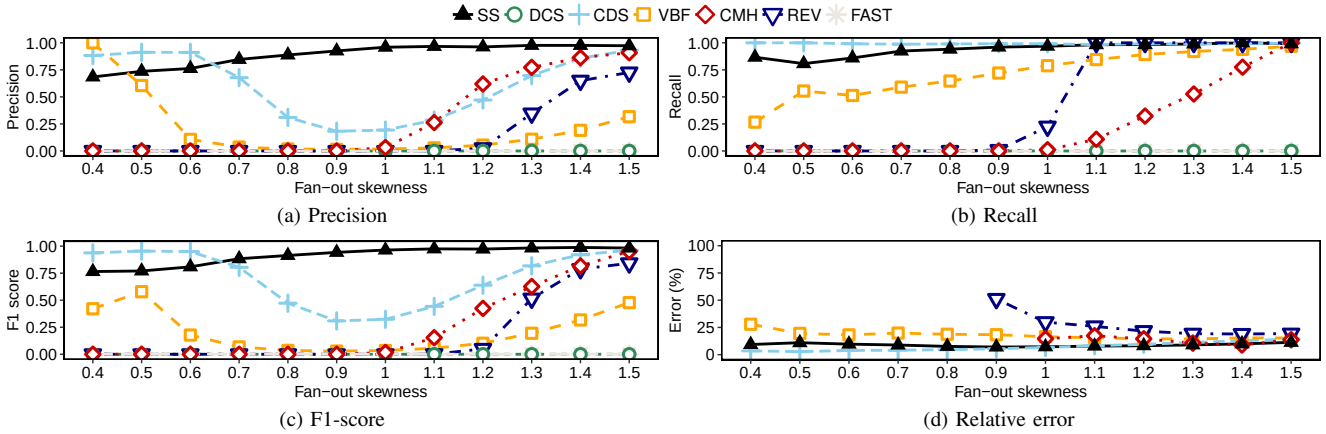


(a) Precision for CAIDA16 (b) Recall for CAIDA16 (c) F1-score for CAIDA16 (d) Relative error for CAIDA16

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

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

Fig. 6. (Experiment 1) Accuracy on CAIDA traces. We do not plot the relative errors for the settings with a zero recall.



(a) Precision (b) Recall

(c) F1-score (d) Relative error

Fig. 7. (Experiment 1) Accuracy on synthetic traces. We do not plot the relative errors for the settings with a zero recall.

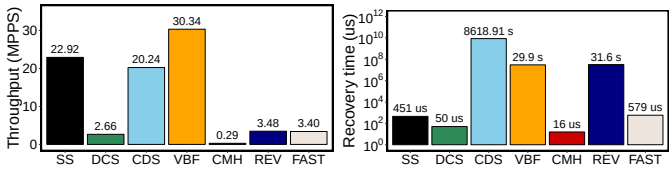
of sources with very large fan-outs. We fix the threshold as 0.001 and the memory as 2 MiB.

Figure 7 shows the results versus the skewness parameter. SpreadSketch maintains stable accuracy and has the highest F1-score in most cases. Its F1-score is over 0.75 in all cases and increases with the skewness parameter. CDS has a recall higher than 0.98, yet its precision drops below 0.2 when the skewness parameter is 0.9. The reason is that at that point, there exist many sources with large fan-outs that are within the threshold, so CDS cannot effectively remove such sources, which lead to a high false positive rate. VBF, CMH, and REV all have an F1-score below 0.5 for the skewness smaller than 1.3. DCS and FAST fail to detect any superspreader in all cases (i.e., zero recall).

(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 for 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.

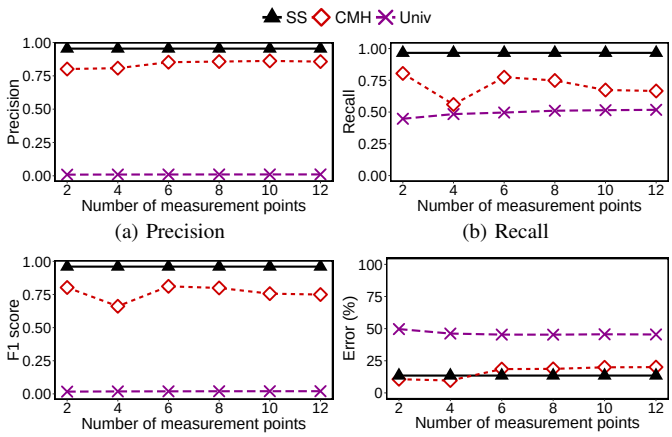
Figure 8(a) shows the update throughput of adding the source-destination pairs of a packet stream into a sketch (in million packets per second (MPPS)). SpreadSketch, CDS, and VBF all achieve throughput above 14.88 MPPS, implying that they can match the 10 Gb/s line-rate in software. VBF has the highest 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. In particular, CMH has the lowest throughput, as it performs fan-out estimation for each packet update (Section II-C). We find that CMH has less than 4% of packets traversing the heap structure, so its major overhead is mainly on fan-out estimation.

Figure 8(b) shows the detection time of returning all



(a) Update throughput (b) Detection time

Fig. 8. (Experiment 2) Performance.



(a) Precision (b) Recall

(a) F1-score (b) Relative error

Fig. 9. (Experiment 3) Network-wide detection.

superspreaders. SpreadSketch returns superspreaders in few milliseconds, while DCS, CMH and FAST return in microseconds, yet the difference is not that significant in practice compared with the epoch length of one minute. 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 [41] 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 for 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.

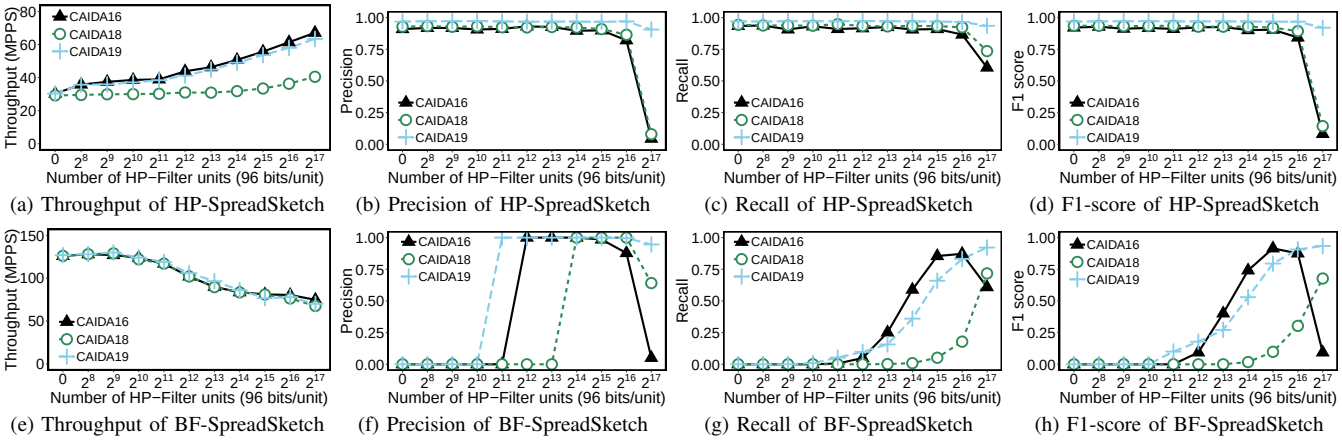
Figure 9 shows the accuracy versus the number of measurement points. The accuracy of SpreadSketch is maintained regardless of the number of measurement points. In contrast, the F1-score of CMH varies with the number of measurement points and generally shows a downtrend. The reason is that CMH only keeps (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 superspreaders 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.

(Experiment 4) Update throughput of HP-SpreadSketch. We study the update throughput gain of HP-SpreadSketch. We fix the total memory sizes of both HP-SpreadSketch and SpreadSketch as 2 MiB. In HP-SpreadSketch, we vary the number of units in the HP-Filter from 0 to 2^{17}, where each unit takes 96 bits (i.e., a 64-bit source-destination pair and a 32-bit counter). When the number of units is zero, it refers to SpreadSketch without the HP-Filter. For SpreadSketch, we fix $r = 4$ rows and change the number of buckets per row (i.e., w), so that the total memory size is 2 MiB (Section VI-A).

Figure 10(a) first shows the update throughput of HP-SpreadSketch versus the number of units in the HP-Filter. The throughput increases with the number of units in the HP-Filter units for all traces. With 2^{13} and 2^{15} units in the HP-Filter (i.e., 4.69% and 18.75% of the total memory size 2 MiB, respectively), HP-SpreadSketch increases the update throughput of SpreadSketch by up to 52.8% and 83.8%, respectively, with no accuracy degradation. The throughput gain is more notable for CAIDA16 and CAIDA19, as they are more skewed than CAIDA18 (Figure 1 in Section III-A). Figures 10(b)-10(d) show the precision, recall, and F1-score of HP-SpreadSketch, respectively. Given the same memory size, HP-SpreadSketch maintains almost the same accuracy as SpreadSketch when the HP-Filter contains no more than 2^{15} units. However, the accuracy of HP-SpreadSketch drops when the HP-Filter has at least 2^{16} units, as less memory is allocated for SpreadSketch. Given the memory constraint, we suggest the HP-Filter should be allocated with no more than 15% of the total memory usage of HP-SpreadSketch.

We also compare HP-SpreadSketch with a variant where the Bloom filter is used instead of the HP-Filter (i.e., BF-SpreadSketch). For comparison purposes, we configure the memory usage of Bloom filter in units of 96 bits (i.e., the size of one unit in the HP-Filter). We vary the memory size of the Bloom filter from 0 to 2^{17} 96-bit units. Figures 10(e)-10(h) show the throughput, precision, recall, and F1-score of BF-SpreadSketch, respectively. Compared with HP-SpreadSketch, BF-SpreadSketch has higher update throughput for all traces, yet its F1-score is below 0.25 for CAIDA18 in most cases. The reason is that for a small memory size, the Bloom filter is almost saturated (i.e., all bits are set) and mistakenly classifies new pairs as repeating pairs that will not be processed by SpreadSketch. Thus, BF-SpreadSketch has almost the same update cost as the Bloom filter, but its accuracy significantly degrades.

(Experiment 5) Update throughput of HP-SpreadSketch versus frequency skewness. Based on the setting in Experiment 4, we further evaluate the update throughput of HP-SpreadSketch by varying the skewness in the frequency of the pairs in different traces (note that we consider the skewness in fan-outs in Experiment 1, while we consider the skewness in frequency here). For each CAIDA trace, we extract all pairs and generate a set of synthetic traces. In each synthetic trace, the number of packets belonging to each pair (i.e., the



(a) Throughput of HP-SpreadSketch (b) Precision of HP-SpreadSketch (c) Recall of HP-SpreadSketch (d) F1-score of HP-SpreadSketch

(e) Throughput of BF-SpreadSketch (f) Precision of BF-SpreadSketch (g) Recall of BF-SpreadSketch (h) F1-score of BF-SpreadSketch

Fig. 10. (Experiment 4) Update throughput of HP-SpreadSketch.

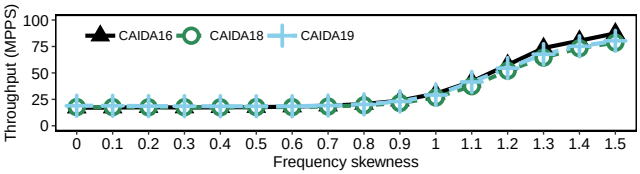
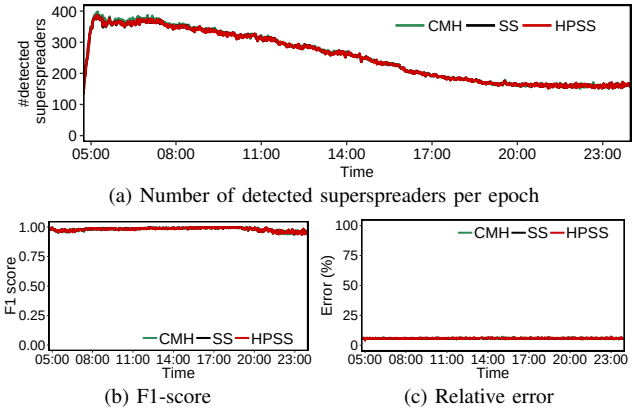


Fig. 11. (Experiment 5) Update throughput of HP-SpreadSketch versus frequency skewness.



(a) Number of detected superspreaders per epoch

(b) F1-score (c) Relative error

Fig. 12. (Experiment 6) Witty worm detection.

frequency of each pair) follows a Zipf distribution with some skewness parameter. A larger skewness parameter means a more skewed distribution; when the skewness parameter is zero, the distribution reduces to a uniform distribution (i.e., each pair in the trace has exactly the same frequency). We fix the total memory usage as 2 MiB and set the number of units in the HP-Filter as 2,048.

Figure 11 shows that HP-SpreadSketch keeps the throughput above 17 MPPS for all cases. The throughput of HP-SpreadSketch increases significantly when the skewness in pair frequency is above 0.9.

(Experiment 6) Real-world attack detection. We study the effectiveness of SpreadSketch in real-world attack detection. We first consider worm detection using the Witty worm trace [5]. We fix the threshold as 0.001 and the memory size of SpreadSketch as 0.5 MiB. We also plot the results of HP-SpreadSketch (HPSS) and CMH. Note that we do not plot the results for other sketches (i.e., DCS, CDS, VBF, REV, and FAST), as their F1-scores are below 0.45.

Figure 12 shows the results, and we first consider SpreadSketch. Figure 12(a) shows that the number of superspreaders that are detected by SpreadSketch in each epoch rises suddenly at time 04:46:00, which is also the actual start time of the attack. We further study the accuracy by comparing the detected superspreaders with the ground truth in each epoch. Figure 12(b) shows that SpreadSketch can track the superspreaders with an F1-score at least 0.96. Figure 12(c) shows that the average relative error of the estimation about the fan-out of each superspreader in SpreadSketch is 0.056. Finally, we observe that both HPSS and CMH have almost identical results to SpreadSketch, yet CMH has much lower throughput than SpreadSketch (Experiment 2).

We also consider the DDoS attack detection using the CAIDA DDoS attack trace [3]. As the trace contains only the attack traffic, we focus on the sub-trace in between the 23-th and 32-th minutes and generate a synthetic DDoS attack trace based on the sub-trace. In particular, we mix the sub-trace with the first ten minutes of the CAIDA16 trace on a per-minute basis. Note that the DDoS attack happens in the middle of the sub-trace. We set the threshold as 3,000 and monitor all hosts that have the number of distinct connections above the threshold.

Figure 13 shows the number of distinct connections of each detected superspreader under SpreadSketch. SpreadSketch reports all four superspreaders (denoted by Hosts A, B, C, and D). Hosts A, B, and D keep a relatively stable and large number of connections over time, while Host C has a surge in the number of connections at the sixth minute. It indicates that Hosts A, B, and D may represent hotspots, while Host C is under a DDoS attack since the sixth minute. The conclusion is consistent with the CAIDA DDoS trace where Host C is indeed the victim and the DDoS attack starts at the sixth minute. We also apply other sketches in DDoS detection. Given the same memory size, HPSS, CMH, REV, and VBF (but not DCS, CDS and FAST) can also detect Host C at the sixth minute (we omit the presentation of the results here).

(Experiment 7) SpreadSketch in hardware. We implement SpreadSketch in P4 [4] (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

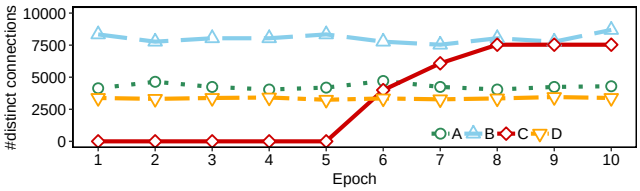


Fig. 13. (Experiment 6) DDoS attack detection.

TABLE III
(EXPERIMENT 7) SWITCH RESOURCE USAGE (PERCENTAGES IN BRACKETS ARE FRACTIONS OF TOTAL RESOURCE USAGE).

SpreadSketch				
SRAM (KiB)	No. stages	No. actions	No. ALUs	PHV size (bytes)
256 (1.67%)	6 (50%)	20 (nil)	6 (12.5%)	108 (14%)
HP-SpreadSketch				
SRAM (KiB)	No. stages	No. actions	No. ALUs	PHV size (bytes)
304 (1.98%)	8 (66.7%)	23 (nil)	7 (14.6%)	110 (14.3%)

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 III 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.

We also implement HP-SpreadSketch in P4 and run it on the Tofino switch, so as to demonstrate the feasibility of deploying HP-SpreadSketch in hardware. We set $t = 256$ units for the HP-Filter and keep the same configuration as above for SpreadSketch. Table III shows that HP-SpreadSketch only slightly increases the hardware resource usage over SpreadSketch. Since SpreadSketch can already process packets at line-rate on our switch, HP-SpreadSketch does not further increase the throughput over SpreadSketch. Nevertheless, we conjecture that HP-SpreadSketch can show its performance benefits as faster switch ASICs evolve in the future.

VII. CONCLUSIONS

Network-wide superspreader detection is a critical task in network management and attack prevention in production networks. 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. To improve the update performance, we extend SpreadSketch with a fast and small data structure, called the HP-Filter, to filter out the heavy pairs and prevent them from being further processed by SpreadSketch. We further implement SpreadSketch in P4 and demonstrate its feasible deployment in commodity hardware switches.

REFERENCES

[1] Barefoot's Tofino. https://barefootnetworks.com/products/brief-tofino/.
[2] CAIDA. http://www.caida.org/data/passive/trace_stats/.
[3] The CAIDA UCSD "DDoS Attack 2007" dataset. https://www.caida.org/catalog/datasets/ddos-20070804_dataset.
[4] P4 language. https://p4.org.
[5] Witty Internet worm. http://www.caida.org/data/passive/trace_stats/.
[6] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner. Heavy hitters in streams and sliding windows. In *Proc. of IEEE INFOCOM*, 2016.
[7] R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik. Faster and more accurate measurement through additive-error counters. In *Proc. of IEEE INFOCOM*, pages 1251–1260. IEEE, 2020.
[8] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proc. of the ACM SIGMOD*, pages 199–210, 2007.
[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *ACM Magazine Communications*, pages 422–426, 1970.
[10] R. S. Boyer and J. S. Moore. MJRTY - a fast majority vote algorithm. In *Automated Reasoning*, pages 105–117. Springer, 1991.
[11] J. Cao, Y. Jin, A. Chen, T. Bu, and Z.-L. Zhang. Identifying high cardinality internet hosts. In *Proc. of IEEE INFOCOM*, 2009.
[12] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. BeauCoup: Answering many network traffic queries, one memory update at a time. In *Proc. of ACM SIGCOMM*, pages 226–239, 2020.
[13] 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.
[14] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *Proc. of ACM PODS*, 2005.
[15] Z. Durumeric, M. Bailey, and J. A. Halderman. An Internet-wide view of Internet-wide scanning. In *Proc. of USENIX Security Symposium*, 2014.
[16] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. on Computer Systems*, 21(3):270–313, 2003.
[17] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proc. of ACM IMC*, 2003.
[18] 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.
[19] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic DDoS defense. In *Proc. of USENIX Security Symposium*, 2015.
[20] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms*, 2007.
[21] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.
[22] D. S. for IMC 2010 Data Center Measurement. https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
[23] 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.
[24] 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.

[25] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity ethernet at acale. In *Proc. of ACM SIGCOMM*, pages 202–215, 2016.
[26] 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.
[27] 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.
[28] 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.
[29] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network monitoring using traffic dispersion graphs (TDGs). In *Proc. of ACM IMC*, 2007.
[30] P. Jia, P. Wang, Y. Zhang, X. Zhang, J. Tao, J. Ding, X. Guan, and D. Towsley. Accurately estimating user cardinalities and detecting super spreaders over time. *IEEE Trans. on Knowledge and Data Engineering*, 2020.
[31] X. Jing, Z. Yan, H. Han, and W. Pedrycz. ExtendedSketch: Fusing network traffic for super host identification with a memory efficient sketch. *IEEE Transactions on Dependable and Secure Computing*, 2021.
[32] N. Kamiyama, T. Mori, and R. Kawahara. Simple and adaptive identification of superspreaders by flow sampling. In *Proc. of IEEE INFOCOM*, 2007.
[33] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proc. of ACM PODS*, 2010.
[34] D. E. Knuth. *The Art of Computer Programming, Volume 4*. Addison-Wesley Professional, 2015.
[35] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek, and G. Antichi. Enabling event-triggered data plane monitoring. In *Proc. of ACM SOSR*, pages 14–26, 2020.
[36] 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.
[37] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *Proc. of USENIX NSDI*, 2016.
[38] 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.
[39] Y. Liu, W. Chen, and Y. Guan. A fast sketch for aggregate queries over high-speed network traffic. In *Proc. of IEEE INFOCOM*, 2012.
[40] 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.
[41] 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.
[42] R. Martin. Snort: Lightweight intrusion detection for networks. In *Proc. of USENIX LISA*, 1999.
[43] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. SCREAM: Sketch resource allocation for software-defined measurement. In *Proc. of ACM CoNEXT*, 2015.
[44] D. Plonka. FlowScan: A network traffic flow reporting and visualization tool. In *Proc. of USENIX LISA*, 2000.
[45] L. E. T. Project. http://www.icir.org/enterprise-tracing/.
[46] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. Dinda, M. Y. Kao, and G. Memik. Reversible sketches: Enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Trans. on Networking*, 15(5):1059–1072, 2007.
[47] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Proc. of ACM SIGCOMM*, 2002.
[48] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proc. of USENIX OSDI*, 2004.
[49] Y.-E. Sun, H. Huang, C. Ma, S. Chen, Y. Du, and Q. Xiao. Online spread estimation with non-duplicate sampling. In *Proc. of IEEE INFOCOM*, pages 2440–2448. IEEE, 2020.
[50] L. Tang, Q. Huang, and P. P. Lee. A fast and compact invertible sketch for network-wide heavy flow detection. *IEEE/ACM Trans. on Networking*, 28(5):2350–2363, 2020.
[51] L. Tang, Q. Huang, and P. P. Lee. SpreadSketch: Toward invertible and network-wide detection of superspreaders. In *Proc. of IEEE INFOCOM*, pages 1608–1617. IEEE, 2020.
[52] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *Proc. of NDSS*, 2005.
[53] 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.
[54] S. Wang, C. Sun, Z. Meng, M. Wang, J. Cao, M. Xu, J. Bi, Q. Huang, M. Moshref, T. Yang, et al. Martini: Bridging the gap between network measurement and control using switching ASICs. In *Proc. of IEEE ICNP*, pages 1–12. IEEE, 2020.
[55] 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.
[56] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen. Estimating the persistent spreads in high-speed networks. In *Proc. of IEEE ICNP*, 2014.
[57] 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.
[58] M. Yoon and S. Chen. Detecting stealthy spreaders by random aging streaming filters. *IEICE Trans. on communications*, 94(8):2274–2281, 2011.
[59] 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.
[60] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *Proc. of USENIX NSDI*, 2013.
[61] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of Internet flow rates. In *Proc. of ACM SIGCOMM*, 2002.
[62] Q. Zhao, A. Kumar, and J. Xu. Joint data streaming and sampling techniques for detection of super sources. In *Proc. of ACM SIGCOMM*, 2005.
[63] Y. Zhou, D. Zhang, K. Gao, C. Sun, J. Cao, Y. Wang, M. Xu, and J. Wu. Newton: Intent-driven network traffic monitoring. In *Proc. of ACM CoNEXT*, pages 295–308, 2020.
[64] 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.

Lu Tang received the Ph.D. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2020. She is now an Assistant Professor of the Department of Computer Science and Technology at Xiamen University. Her research interests are in network measurement and data center networks.

Yao Xiao is pursuing his Master degree in Computer Science and Technology at Xiamen University. His research interests are in network measurement and sketch algorithms.

Qun Huang received the Ph.D. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2015. He is now an Assistant Professor of the Department of Computer Science and Technology at Peking University. His research interests are in distributed stream processing and network measurement.

Patrick P. C. Lee received the Ph.D. degree in Computer Science from Columbia University in 2008. He is now a Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including storage systems, distributed systems and networks, and cloud computing.