Super Spreader Identification Using Geometric-Min Filter

Chaoyi Ma, *Graduate Student Member, IEEE*, Shigang Chen, *Fellow, IEEE*, Youlin Zhang, Qingjun Xiao, *Member, IEEE, ACM*, and Olufemi O. Odegbile

***Abstract*— Super spreader identification has a lot of applications in network management and security monitoring. It is a more difficult problem than heavy hitter identification because flow spread is harder to measure than flow size due to the requirement of duplicate removal. The prior work either incurs heavy memory overhead or requires heavy computations. This paper designs a new super-spreader monitor capable of identifying all flows whose spreads are greater than a user-specified threshold with a probability that can be arbitrarily set. It introduces a generalized geometric hash function, a generalized geometric counter, and a novel geometric-min filter that blocks out the vast majority of small/medium flows from being tracked, allowing us to focus on a small number of flows in which super spreaders are identified. We provide an analytical way of properly setting the system threshold to meet probabilistically guaranteed identification of super spreaders, and implement it on both hardware (FPGA) and software platforms. We perform extensive experiments based on real Internet traffic traces from CAIDA. The results show that with proper parameter settings, the new monitor can identify more than 99% super spreaders with a low memory requirement, better than the prior art.**

***Index Terms*— Traffic measurement, super spreader identification.**

I. Introduction

TRAFFIC measurement provides critical information for network management and security monitoring. While the explosive growth in both router speed and traffic volume places ever increasing strain on faster packet processing on the line cards, the demand of sophisticated network management only adds more to the stress, requiring traffic measurement tasks to operate efficiently and compete less for the limited on-chip resources such as SRAM on the data plane of routers and other network devices [1].

A packet stream under monitoring is modeled as a set of flows. Each flow consists of all packets carried the same flow label, which is usually a combination of one or multiple fields in the packet header. For individual flows, there are two basic types of measurements: size and spread. The *size* of a flow is the number of elements carried in the packets of the flow, where elements may refer to whole packets, bytes in payload, or certain content. The spread of a flow is the number of *distinct* elements in the flow, where elements may refer to addresses, ports, other header fields, or payload content. The difference between the two is that measuring a flow's spread requires us to remove duplicate elements, whereas measuring a flow's size does not. Consider an example where all packets sent to the same destination constitutes a flow. For flow size, suppose that the network admin wants to measure the number of packets, where each packet is treated as an element. For flow spread, suppose that the admin wants to measure the number of distinct source addresses carried in the packets. Let each measurement period be 1 minute. Consider the case that a single source sends 1,000,000 packets to a server. This flow's size is clearly 1,000,000, while its spread is just 1. Now consider a different case that 1,000,000 packets come from 100,000 different sources, then the size is still 1,000,000, but the spread is 100,000. If the server normally receives requests from 1,000 sources, the abnormal spread of 100,000 should raise an alert (e.g., possible DDoS attack).

Of particular importance are *heavy hitters* with large flow sizes and *super spreaders* with large flow spreads. Heavy hitter identification has received intensive research [2]–[11], and it has many applications in router/switch configuration [12], load balancing [13], and routing optimization [14], [15], etc. The focus of this paper is on super spreader identification, which is a more challenging problem due to the difficulty of duplicate removal. It has wide applications in anomaly detection [16], [17], Internet search trend detection [18], DDoS attack detection [19], user/content profiling [20], etc. For another example, data analysis systems at Google such as PowerDrill [21] and Dremel [22] measure number of distinct search over a time period [23]. By measuring distinct searches (spread) for each subnetwork and identifying those subnetworks with high spreads (super spreaders), it may forward the requests from them to powerful machines in its server farm to improve performance, while forwarding requests from low-spread subnetwork to other machines.

Before we discuss the state of the art in super spreader identification, we address a related problem: per-flow spread estimation [23]–[32], which provides an estimated spread for each flow. It may appear that a solution for per-flow spread estimation can be used to identify super spreaders. This is actually not true if one wants an efficient solution that operates as a compact on-chip module to process packets at line rate. To fit in small space, most (if not all) existing solutions for per-flow spread estimation do not keep flow labels. Their query overhead is too large to perform per-packet spread query. These two factors prevent them from being used for super

Manuscript received September 8, 2020; revised April 19, 2021; accepted August 20, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor R. Lo Cigno. This work was supported by NSF under Grant CNS-1719222. *(Corresponding author: Chaoyi Ma.)*

Chaoyi Ma, Shigang Chen, and Youlin Zhang are with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: ch.ma@ufl.edu; sgchen@cise.ufl.edu; ylzh10@ufl.edu).

Qingjun Xiao is with the School of Cyber Science and Engineering, Southeast University, Nanjing, Jiangsu 211189, China (e-mail: csqjxiao@seu.edu.cn).

Olufemi O. Odegbile is with the Department of Computer Science, Clark University, Worcester, MA 01610 USA (e-mail: oodegbile@ufl.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNET.2021.3108033, provided by the authors.

Digital Object Identifier 10.1109/TNET.2021.3108033

spreader identification. Existing solutions that are specifically designed for super spreader identification include [33]–[38], which will be explained in details shortly. Among them, AROMA [35] and SpreadSketch [38] are the state of the art. However, SpreadSketch still relies on a memory-heavy data structure that supports per-flow spread estimation. AROMA cannot deal well with flows of very large spreads.

In this paper, we design a new super spreader identification solution that outputs a list of flows whose spreads are greater than a user-specified threshold at the end of each measurement period. It introduces a generalized geometric hash function, a generalized geometric counter, and a novel geometric-min filter that not only removes duplicates but also blocks out the vast majority of small/medium flows. Only a small number of flows that pass the filter will be tracked with a hash table. After a measurement period, we measure the spreads of these flows and identify the super spreaders among them. We provide an analytical way of calculating the optimal parameter settings to meet probabilistically guaranteed identification of super spreaders. We generalize our solution to network-wide measurement. We implement the proposed solution on both hardware (FPGA) and software platforms. We perform extensive experiments based on real Internet traffic traces from CAIDA [39]. The results show that our new solution outperforms the best prior art in efficiently and accurately identifying super spreaders.

The rest of this paper is organized as follows. Section II introduces the problem we explore in this paper. Section III discusses the prior work and provides technical motivation. Section IV gives a description of generalized geometric hash and geometric counter. Section V uses geometric counters to form a geometric-min filter for super spreader identification. We evaluate the performance of our algorithm using both software and hardware implementations in Section VI before drawing the conclusion in Section VII.

II. System Model and Problem Statement

A. System Model

A network traffic measurement system consists of an online recording module and an offline processing module. The online module is to monitor the network traffic and a router, switch, gateway, intrusion detection system(IDS) or other network device. To keep up with the line rate, we expect the module to co-locate with other per-packet processing functions such as forwarding, access Control List(ACL), packet inspection, etc., which are often implemented on network processing chip using on-chip cache memory for high speed. It is highly desirable for the online module to be light-weight in both memory footprint and processing overhead so as to leave more resources to other network functions. The offline module can be more time-consuming and space-consuming because it is not under real-time constraints and can be left to a powerful server. We divide time into measurement periods whose length will be determined based on application need. During each period, the online module (running on a network device) will extract information of interest from the arrival packets and update its data structures. At the end of the period, the network device will offload its data to a server and then reset its data structures to restart a new period. After receiving the data, the server will perform traffic measurement computation for a given purpose such as identifying super spreaders.

TABLE I
Explanation of Some Notations

f	flow identifier
e	element identifier
b	base of a geometric counter
$G_b(.)$	generalized geometric hash function of base b
$H(.)$	uniform hash function
t	length of a geometric counter
g_b	geometric counter of base b
d	number of g-counter arrays
l	length of each g-counter array
$F_i[j], 0 \le i \le d-1, 0 \le j \le l-1$	jth g-counter of ith array
n_f	spread of flow f
U	user-defined spread threshold
α	detection rate bound
$H_i(.), 0 \le i \le d-1$	uniform hash function for ith g-counter array
$G_{b,i}(.), 0 \le i \le d-1$	geometric hash function for ith g-counter array with base b
T	geometric counter threshold
$v(f)$	geometric-min value of flow f
CDS	compact data structure for per-flow spread measurement

B. Problem Statement

We abstract each arrival packet as a pair $\langle f, e\rangle$, where f is a flow label and e is an element identifier. Flows and elements under monitoring can be arbitrarily defined based on different application needs. Flow identifier may be chosen based on packet-header fields or application-header fields such as source address, destination address, protocol, and/or port numbers. Elements may also be chosen from the packet header, application header or even payload. The spread of a flow f, denoted as n_f, is the number of distinct elements in all arrival packets whose flow identifier is f during a certain measurement period.

The problem we investigate in this paper is called *super spreader identification*, which is to report and measure the flows whose spreads exceed a user-defined spread threshold U. In other words, a flow f is a *super spreader* if $n_f \ge U$. We want to ensure that the probability of reporting a flow whose true spread is at least U will be no less than α, i.e.,

$$Prob\{\text{report}f | n_f \ge U\} \ge \alpha, \tag{1}$$

where α is a user-defined probability. Under the above requirement, our solution design will also need to control the false positive rate (which happens when a flow with $n_f < U$ is mistakenly reported) to a low level. Notations used in this paper are given in Table I for quick reference.

III. Prior Art and Motivation

A. Per-Flow Measurement

Much prior work is to measure per-flow size [8], [29], [40]–[50], which counts the number of elements in each flow, where elements may be packets, bytes or occurrences for certain events or content in packets. It takes a counter to measure the size of each flow. With each arrival packet, the counter is increased by an integer (such as one for packet count). Measuring per-flow spread is more difficult [26], [29], [30], [51]. With each arrival packet $\langle f, e\rangle$, we have to first determine whether e has appeared before. It requires

a more sophisticated data structure such as bitmap [28], [52], FM sketch [53] or HLL sketch [23], [54], [55], each taking hundreds or thousands of bits to record the elements that have been seen. Per-flow measurement cannot be used directly for heavy hitter/super spreader identification since they never store the information of flow labels.

B. Super Spreaders vs. Heavy Hitters

The problem of *heavy-hitter identification* is closely related to our study, which is to find flows of large sizes. The most prevalent approach is to keep track of a small set of flows, trying to retain large flows in the set while replacing small ones with new flows. Prior solutions include Frequent [6], Lossy Counting [5], Space Saving [4], CSS [3], RHHH [56], Heavy Keeper [2], SketchLearner [57], Elastic Sketch [8] and Nitrosketch [58]. The data structures for keeping a small set of flows include min-heap [59], stream summary [4], TinyTable [60], and hash table [8]. Sampling also helps filter small flows [58].

We explain two solutions as examples. Space Saving [4] keeps a number of flows and their sizes in a stream summary, which takes $O(1)$ time to update the size of any flow or find the smallest flow. When a packet of a flow in the summary arrives, the flow's counter is increased by one. When a packet of a new flow arrives, we replace the smallest flow f in the summary with the new one whose size is initialized to the size of f plus one. Heavy Keeper [2] uses hash tables. Each hash entry stores one flow's ID and a counter. When a packet of a flow in the table arrives, the flow's counter is increased by one. When a packet of a new flow arrives, if the flow is hashed to an entry already having a flow, that flow's counter is decayed. When the counter is decayed to zero, a new flow will replace it.

None of the solutions for heavy hitter identification can be used for super spreader identification since they use counters. A counter cannot keep track of a flow's spread, i.e., the number of distinct elements in the flow. As we explained earlier, this requires a data structure [23], [28], [53], [55] that can "remember" what elements have been seen.

One might argue to still use the heavy-hitter solutions but replace their counters with some other data structures [23], [28], [53], [55]. There are two problems. The first one is that these data structures take a lot more space than counters. For example, each HLL sketch [55] will need over a thousand bits for good accuracy, compared to 32 bits of a counter. Such space "inefficiency" means that we need to limit the number of flows that are monitored. All heavy hitter (super spreader) solutions track a subset of flows that include hopefully all heavy hitters (super spreaders) and inevitably some other flows. Given a certain space constraint, the more memory each flow will use, the fewer the number of flows we can track.

The second problem is more serious. The heavy-hitter solutions all require online queries of flow sizes, even on a per-packet basis. For example, when receiving the packet of a new flow (which is not tracked in the stream summary), Space Saving [4] has to find the smallest flow size currently in the summary for replacement operation. There are numerous small flows in typical network traffic, and the arrival packet stream may contain a long sequence of new-flow packets. While accessing a counter for flow size is cheap, it is very expensive to compute a flow's spread from HLL or other sketches [28], [53], [55] that encode a large number of distinct elements. Our experiment shows that it is at least four orders of magnitude slower to compute a flow's spread online from HLL than to find a flow's size from a counter.

Therefore, for online efficiency, we decide that our solution for super spreader identification should not include any per-packet operation that needs to compute any flow spread. This requirement excludes the approaches in the heavy-hitter solutions from being considered in this paper. We need a different solution structure.

C. Existing Solutions

We discuss the existing solutions for super spreader identification and their problem here.

Venkataraman *et al.* [33] use sampling and a hash table to store the flow identifiers (source addresses) and, for each flow, uses a hash table (or a Bloom filter) to record the distinct elements of the flow. DCS [34] uses multiple hash tables, each with a different sampling probability, to store $\langle f, e\rangle$ pairs, from which we can count the number of distinct sampled elements from each flow in each table, produce an estimate adjusted by sampling probability, select the most accurate estimate from different hash tables, and identify super spreaders. By actually storing the element identifiers (or encoding them in Bloom filters), the memory overhead is very large.

FAST [37] maintains multiple arrays of HLL sketches [55]. For each arrival packet $\langle f, e\rangle$, it splits f into two parts, hashes one part to d HLL arrays, and in each array records e in one HLL sketch for every bit in the second part of f whose value is one. Without storing element identifiers, this method uses less memory than [33], [55]. However, it has to record element e many times in each of the d arrays. This still causes significant memory overhead and inaccuracy as each HLL sketch has to be shared by many flows. Moreover, it is very computationally expensive to recover the flow identifiers. We take a new approach that is different from the above solutions. We rely on a light-weight filter to separate super spreaders from the vast majority of small flows. Not only does such an approach drastically reduce memory usage, but it increases the accuracy in super spreader identification and their spread measurements.

CMH [36] uses Count-Min and a min-heap whose counters are replaced by a data structure such as bitmap that can measure flow spread. As we explained earlier, this method has significant memory overhead. For each arrival packet $\langle f, e\rangle$, while recording the element, it queries the spread of flow f. If the estimated spread of f is larger than a threshold, it will report f as a super spreader. However, spread estimation is computationally expensive and not suitable for online per-packet operation.

UnivMon [7] is designed to measure per-flow size and the number of different flows (called the cardinality) in a packet stream. Using the term of spread in this paper, UnivMon's cardinality measurement can be logically considered as measuring the spread of a single flow if we treat the whole packet stream as a flow. For instance, if we only measure the packets to a single destination address X, UnivMon can be used to measure the number of distinct source addresses that have sent packets to X for DDoS attack detection [7]. However, if we want to monitor many destination addresses simultaneously for DDoS attacks, we would need one UnivMon data structure for each destination or replace each counter in UnivMon with a multi-resolution bitmap [61], which would be very costly in memory consumption and hurt performance, as is observed in [38].

SpreadSketch [38] can measure the spreads of many flows simultaneously and identify the super spreaders among them. Its data structure follows Count-Min [43] but replaces each counter with a multi-resolution bitmap [61], a register and a label field. Multi-resolution bitmap is designed to measure the spread of one flow. Though it takes a considerable amount of memory, it is more space-compact than UnivMon for spread measurement. The label field is used to record one flow label. The experiments in [38] show that if we replace each counter in UnivMon with a multi-resolution bitmap, the performance is much worse than SpreadSketch. While Count-Min [43] is generally considered to be a very compact data structure, that is not necessarily true for SpreadSketch after each 32-bit counter is replaced with a multi-resolution bitmap of 438 bits, a register of 3 bits, and a label field of 32 bits. Our observation is that the design of SpreadSketch is overprovisioned: With numerous multi-resolution bitmaps placed in a Count-Min structure, it is able to, unnecessarily, provide a spread estimate for any flow, even though the final task is only to identify super spreaders. Its use of per-flow spread estimation carries the cost of high memory overhead. Our idea is to avoid a heavy-duty data structure of per-flow spread estimation, but to create a light-weight filter that blocks out most small flows, so that we only need to measure the spreads of a small number of post-filter flows. This strategy will allow us to save memory over SpreadSketch or, equivalently, improve accuracy under the same memory allocation.

AROMA [35] adopts a sampling strategy. It allocates a bucket array where each bucket stores a flow label and a counter. It first hashes each packet $\langle f, e\rangle$ into a bucket in the array and then produces another hash value of $\langle f, e\rangle$. If the hash value is smaller than the counter of the bucket, the counter is set to this value and the flow label is set to f. Because a flow of higher spread has more distinct elements, they together are more likely to produce small hash values, meaning that more of them will likely stay in the array. Hence, we can identify super spreaders by finding the flow labels that appear most in the array, and estimate their spreads based on their counts in the array. Storing the same flow labels many times in the array can cost significant memory overhead, especially when the flow labels are long (such as 104 bits for each TCP flow label). Moreover, when there exists a flow of very large spread, it could push other super spreaders of smaller spreads out of the array, causing either failure in identifying some super spreaders or inaccuracy in their spread estimations, as our experiments will demonstrate.

IV. Geometric Counter

In this section, we introduce a generalized geometric hash function and a generalized geometric counter to lay the foundation for our solution to the problem of super spreader identification.

A. Generalized Geometric Hash Function

We define a *generalized geometric hash function* with base b as follows: Given an input e, its value (denoted as $G_b(e)$) has the following distribution,

$$Prob\{G_b(e) = k\} = (\frac{b-1}{b})^k \frac{1}{b}, \tag{2}$$

where k is a non-negative integer. In this paper, we consider the base values that are powers of 2, i.e., $b = 2^w$ with integer $w \geq 1$, because they can be efficiently implemented from a uniform hash function $H(.)$ that produces a pseudo-random output given any input. Consider an arbitrary input e, we divide $H(e)$ into segments of w bits each, where the ith segment is denoted as $S[i] = H(e)[iw, \ldots, ((i+1)w-1)]$, consisting of w consecutive bits from the iwth bit in $H(e)$ to the $((i+1)w-1)$th bit, for $i \geq 0$. The value of $G_b(e)$ is determined as follows:

$$G_b(e) = \begin{cases} 0, & \text{if } S[0] = 0 \\ k, & \text{if } \exists k > 0,\ S[i] \neq 0,\ 0 < i \leq k, \\ & \text{and } S[k] = 0 \end{cases} \tag{3}$$

$G_b(e) = k$ when the kth segment is zero and all previous segments are non-zeros. A segment is zero when all its bits are zeros.

We prove (2). When $k = 0$, the probability for $S[i] = 0$ is $\frac{1}{2^w} = \frac{1}{b}$. When $k > 0$, the probability of $S[i] \neq 0$, $0 < i \leq k$, is $(\frac{b-1}{b})^k$. The probability of $S[k] = 0$ is $\frac{1}{b}$. Hence, the probability of $G_b(e) = k$ is $(\frac{b-1}{b})^k \frac{1}{b}$.

For geometric hash with b not a power of 2, we need to re-define $S[i]$ through two series, $S[0] = H(e) \mod b$, $Q[0] = \lfloor \frac{H(e)}{b} \rfloor$, $S[i] = Q[i-1] \mod b$, and $Q[i] = \lfloor \frac{Q[i-1]}{b} \rfloor$, for $i > 0$. The value of $G_b(e)$ is still determined by (3). Let t be largest output of $G_b(e)$. We need $\lceil \log_2 b^t \rceil = \lceil t \log_2 b \rceil$ bits in $H(e)$. For example, if $t = 32$ and $b = 2$, then we need 32 bits from $H(e)$. If $t = 32$ and $b = 8$, we need 96 bits from $H(e)$. In case that $H(e)$ produces less than that, we can perform an additional hash $H(H(e))$ with $H(e)$ as input. The pseudo code for generalized geometric hash is given in Alg.1.

Algorithm 1 Generalized Geometric Hash

Input: e, b, t
Output: geometric hash value $G_b(e)$

```
1: G_b(e) = 0
2: S[0] = H(e) mod b
3: Q[0] = ⌊H(e)/b⌋
4: for i = 1..2^t − 1 do
5:   if S[i] = 0 then
6:     break
7:   else
8:     G_b(e) = G_b(e) + 1
9:   end if
10:  S[i] = Q[i − 1] mod b
11:  Q[i] = ⌊Q[i − 1]/b⌋
12: end for
13: return G_b(e)
```

We briefly explain that Alg. 1 is equivalent to (3) when b is power of two, e.g., $b = 2^w$. The mod operations at lines 2 and 10 actually extract the lowest w bits of $H(e)$ and $Q[i-1]$, while the divide operations at lines 3 and 11 keep the rest bits. Under this definition, the value of $S[i]$ is the iwth bit to the $((i+1)w-1)$th bit in $H(e)$. This is the same as what we described before (3). Therefore, Alg. 1 is equivalent to (3) when b is power of two. A geometric hash function can also be applied to a pair $\langle f, e\rangle$, denoted as $G_b(f, e)$, which can be implemented by replacing $H(e)$ and $H(f) \oplus H(e)$.

B. Generalized Geometric Counter

We define a *generalized geometric counter* g_b of t bits long with base b, where $r > 1$ and $b > 1$ are two integer parameters chosen by the user. While it is called a counter (specifically geometric counter), it is totally different from the traditional counter that we refer to in Section III. The operation of recording $\langle f, e\rangle$ in a counter g_b is not a simple increment of one, but first performs a geometric hash $G_b[f, e]$ and then sets g_b to this value only if it is greater than the current counter value. That is, $g_b = \max\{g_b, G_b[f, e]\}$.

Thanks to the pseudo-random nature of geometric hashing, a geometric counter automatically removes duplicate elements (which refer to the same $\langle f, e\rangle$ pair that appears again in a packet stream). That is because $G_b[f, e]$ produces the same value no matter how many times $\langle f, e\rangle$ appears. After the first appearance, its duplicates will have no impact on the counter value.

Again due to the pseudo-random nature of geometric hashing, a geometric counter cannot provide an accurate count on the number n of distinct elements recorded in the counter. But it can serve as a probabilistic filter. Consider a geometric counter that after recorded n distinct elements from a flow. For any non-negative integer k, we have

$$Prob\{G_b(f, e) \geq k\} = (\frac{b-1}{b})^k, \tag{4}$$

which happens when $S[i] = 0$, $0 \leq i < k$, each with a probability $\frac{b-1}{b}$. Suppose the initial value of g_b is zero. After recording n elements, we have

$$\begin{aligned} Prob\{g_b \geq k\} &= 1 - (1 - Prob\{G_b(e) \geq k\})^n \\ &= 1 - (1 - (\frac{b-1}{b})^k)^n. \end{aligned} \tag{5}$$

We may use g_b as a filter for a flow f such that the flow is reported if $g_b \geq T$, where T is a geometric counter threshold. We determine T from the requirement (1) as follows:

$$\begin{aligned} Prob\{\text{report } f | n_f \geq U\} &\geq \alpha \\ Prob\{g_b \geq T | n_f \geq U\} &\geq \alpha \\ 1 - (1 - (\frac{b-1}{b})^T)^U &\geq \alpha \\ T &\leq \log_{\frac{b-1}{b}} (1 - (1-\alpha)^{\frac{1}{U}}) \end{aligned}$$

The base b controls a space-accuracy tradeoff. On the one hand, when we increase b, for instance, from 2 to 4 to 8, the number of bits in g_b has to be at least $\log_2 T$, which increases as b decreases. On the other hand, a larger size of g_b can record flow spread at finer granularity, which affects the filter performance.

One may find that when $b = 2$, our geometric counter is actually a HLL register [55]. That is correct but the design purpose of our geometric counter is different. HLL is designed for accurate spread estimation, therefore, it requires tens or hundreds of registers to work. In contrast, several geometric counters are enough to serve the function of blocking the small flows as we will show in Section V.

One problem is that a separate geometric counter for each flow is per-flow information. While the number of super spreaders is typically small, the number of small flows can be huge in a modern high-speed network. It will cause significantly overhead for the online module of traffic measurement to keep every flow's identifier f and its geometric counter in a hash table. The problem will become serious if the online module is implemented in on-chip cache memory to match line rate [29]. Our idea in this paper is to use a fixed number of geometric counters as a combined filter for all flows. This combined filter will block out small flows.

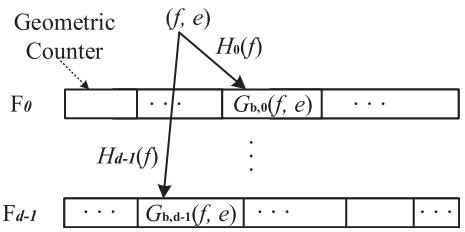


Fig. 1. Geometric-min filter.

V. GEOMETRIC-MIN FILTER

To identify super spreaders, our solution will rely on a geometric-min filter (consisting of multiple geometric counters) that works on all flows to separate super spreaders from others.

A. Main Design

The proposed online module consists of a geometric-min filter, a hash table and an optional part for specific application needs, i.e., a compact data structure that can do per-flow spread measurement (abbreviated as CDS), which we will discuss shortly in this subsection. Our geometric-min filter itself will identify the super spreaders and the hash table is to store the flow identifiers of all flows that pass the filter. With the filter, the requirement (1) is rewritten as

$$Prob\{f \text{ passing filter} | n_f \geq U\} \geq \alpha. \tag{6}$$

For each arrival packet $\langle f, e\rangle$, we hash f to the hash tale and do one of the following:

1) If f is not in the hash table, we insert e into the filter and check whether f can now pass the filter (The certification for passing the filter will be given later). If it does, we insert f into the hash table.
2) If f is in the hash table, the flow has already passed the filter. We do nothing.

At the end of a measurement period, the network device hosting the online module will offload the hash table to a central server where the offline module is implemented. The server will compute the spreads of the flows in the hash table from the compact data structure; note that this is offline operation that is not driven by packet arrival.

In the sequel, we elaborate the design of geometric-min filter, determine its geometric counter threshold in order to satisfy (6), analyze the performance of our solution, and discuss network-wide measurement with geometric-min filter.

As shown in Fig. 1, a geometric-min filter consists of d arrays of geometric counters, denoted as F_i, $0 \leq i < d$. Let l be the number of counters in each array. To distinguish from the traditional counters, we refer to geometric counters as *g-counters*. We denote the jth g-counter in the ith array as $F_i[j]$, $0 \leq j < l$, $0 \leq i < d$. All g-counters are initialized to zeros at the beginning of each measurement period. Along with the arrays, we keep a hash table for recording labels of the passed flows.

Consider an arbitrary packet $\langle f, e\rangle$, if f has already passed the filter and recorded by the hash table, we ignore it; otherwise, we record it into the filter. First, we perform d independent hashes, $H_i(f)$, $0 \le i < d$, locate d g-counters, $F_i[H_i(f)], 0 \le i < d$, one in each array. We then use d independent geometric hash functions of base b, $G_{b,i}, 0 \le i \le d-1$, to record the element in the filter as follows:

$$F_i[H_i(f)] = \max\{F_i[H_i(f)], G_{b,i}(f,e)\}, \quad 0 \le i < d. \quad (7)$$

We define the geometric-min value $v(f)$ of flow f in the filter as

$$v(f) = \min\{F_i[H_i(f)], 0 \le i < d\}. \quad (8)$$

If $v(f) \ge T$, the flow passes the filter and we insert it to the hash table, where T is a geometric counter threshold.

The algorithm for geometric-min filter update is given in Alg. 2. We check the hash table before we update the filter because when f is already in the hash table, we do not need extra hash operations for updating filter. This improves the throughput. We stress that if a flow is already in the hash table, the filtering performance will not be affected whether we keep recording its packets in the filter or not. The reason is that all registers for any flow in the hash table must contain a value larger at least T and updating on those registers will not affect the filtering performance.

Algorithm 2 Geometric-Min Filter

Input: filter F, b, U, α, hash table HT
Action: Do filter

```
T = ⌊log_{(b-1)/b}(1 − (1 − α^{1/d})^{1/U})⌋
for each arrival packet ⟨f, e⟩ do
  if f ∉ HT then
    v(f) = INT_MAX
    for i = 0..d − 1 do
      F_i[H_i(f)] = max{F_i[H_i(f)], G_{i,b}(f, e)}
      v(f) = min{F_i[H_i(f)], v(f)}
    end for
    if v(f) ≥ T then
      Insert f to HT
    end if
  end if
end for
```

The design of the filter seems similar to cSkt(HLL) [29], which replaces the counters in Count-Min with HLL that can produce flow spread estimation in order to measure per-flow spreads. However, they are indeed different for two main reasons.

First, the designing purposes and plug-ins are different. cSkt(HLL) aims to provide accurate flow spread estimation for all flows. Therefore, the plug-in, e.g., HLL [55], can provide accurate spread estimation with some complex querying operations. It does not record any flow labels because it is not its purpose and the querying operations are too complex online. In comparison, our filter aims to provide labels of those possible super spreaders using the geometric counters as the plug-ins. It cannot provide accurate spread estimation but only need simple read operations for querying. This make it suitable for online querying and then track the labels for possible super spreaders. Besides, a HLL takes 640 bits while a g-counter only takes several bits, which means the filter is not memory-expensive.

Second, the design of taking the minimum from d arrays are common in many researches. cSkt(HLL) [29] and many other works [2], [36], [38], [47], [58] records the same information of a flow in d plug-ins, each in one array. They choose the plug-in with minimum value as the estimate since it contains least noise information. It is actually a noise reduction method. However, the minimum operation in our filter is not only for noise reduction. As (7) shows, when we update d g-counters for a flow, we use d independent geometric hash functions $G_{b,i}, 0 \le i \le d-1$. This means the values each packet recorded in its d g-counters are different. It reduces the probability that a small flow passing through the filter because it is hard for a small flow to make the values in all d g-counters larger than the threshold.

We use an additional alongside compact data structures (CDS) for accurate spread estimation. Here are many existing solutions doing per-flow spread estimation. The state-of-the-art is vSkt(HLL), which is adopted in this paper. The CDS itself, similar to cSketches that we discuss before, does not record any flow labels. It can only provide a spread estimate for a given flow labels and thus cannot do super spread identification. At the end of an measurement period, we query all labels in the hash table and output the super spreaders based on the spread estimation from CDS.

B. Setting Threshold T

We determine the threshold T to meet the requirement for geometric-min filter in (6). Due to space limitation, we do not put proofs here. Readers can find the proofs in supplemental materials or the appendix part of this paper' full version [62].

Lemma 1: Consider a g-counter for flow f, $F[H_i(f)], 0 \le i < d$, its recorded distinct elements after recording the last element of f(denoted as N_i) has the following properties:

$$N_i \ge n_f \quad (9)$$

$$Prob\{N_i \ge n_f + \frac{n^*}{l\beta}\} \le \beta \quad (10)$$

with n_f the spread of f, n^* number of distinct elements in all flows, l length of each g-counter array and $0 \le \beta < 1$.

Theorem 1: If the geometric counter threshold T in our geometric min filter satisfies

$$T \le \log_{\frac{b-1}{b}}(1 - (1-\alpha^{\frac{1}{d}})^{\frac{1}{U}}) \quad (11)$$

the filter meets the requirement

$$Prob\{f \text{ passing filter}|\ n_f \ge U\} \ge \alpha \quad (12)$$

with n_f spread of flow f, b the base of g-counter and d the number of arrays in the filter.

Theorem 2: If the geometric counter threshold is T in our geometric-min filter, a flow f with spread less or equal to L will be blocked with at least a probability

$$\max\{1 - (1 - (1 - (\frac{b-1}{b})^T)^{L + \frac{n^*}{l\beta}}(1-\beta))^d\} \quad (13)$$

with n_f spread of flow f, n^* number of distinct elements in all flows, b the base of g-counter, l length of each g-counter array, d the number of arrays in the filter and $0 < \beta < 1$.

From Theorem 2, we know that a larger T will have a higher probability of blocking a small flow (spread less than L). Therefore, in practice, we set $T = \lfloor \log_{\frac{b-1}{b}}(1 - (1-\alpha^{\frac{1}{d}})^{\frac{1}{U}})\rfloor$ as it is the largest T we can set according to Theorem 1.

C. Impact of Other Parameter Settings

In this section, we discuss the impact of parameter settings including l, d and b under given U, α and $n*$.

We present the following theorem to discuss how to setting l and then give an corollary for the space complexity of our filter. Readers can find the proofs in supplemental materials or the appendix part of this paper' full version [62].

Theorem 3: If the geometric counter threshold is set as T in our geometric min filter and we want to guarantee that a flow f with spread less or equal to L will be blocked with a probability at least γ, then we have to set the length of filter array as

$$l \geq \min\{\frac{n^*}{\beta(-L+\log_{1-(\frac{b-1}{b})^T}\frac{1-(1-\gamma)^{\frac{1}{d}}}{1-\beta})}|\beta \in (0,1)\} \quad (14)$$

with n^* the number of distinct elements in all flows, b the base of g-counter, l the length of each g-counter array and d the number of arrays in the filter.

Corollary 1: The space complexity of our geometric min filter is $O(\frac{2n^*d(\log_2(1+\log_{\frac{b-1}{b}}(1-(1-\alpha^{\frac{1}{d}})^{\frac{1}{U}}))+1)}{-L+\log_{1-\frac{b}{b-1}(1-(1-\alpha^{\frac{1}{d}})^{\frac{1}{U}})}2(1-(1-\gamma)^{\frac{1}{d}})})$, if we want to guarantee

$$Prob\{f \text{ passing filter}|\ n_f \geq U\} \geq \alpha$$
$$Prob\{f \text{ blocked by filter}|\ n_f \leq L\} \geq \gamma \quad (15)$$

where n^* is the number of distinct elements in all flows.

From Theorems 2 and 3, the g-counter array length l affects the probability for blocking small flows. Under given U, α and n^*, Theorem 1 guarantees the upper bound of T which has no relationship with l. A larger l make the probability for blocking small flows larger. However, a larger l also means larger memory requirement.

The base b of geometric hash is an integer that is greater than or equal to 2. It controls a three-way tradeoff: First, from equation (11), T increases as b increases. Each g-counter in the filter takes at least $t = \lceil log_2(T+1) \rceil$ bits, which also increases. Hence, a larger value of b means greater memory overhead. Second, according to Section IV, the number of uniform hash operations increases as b increases in order to produce a geometric hash output.

Let $\alpha = 0.99, d = 4, l = 4096$ and $l_{hash} = 64$. Tables II-IV demonstrate the three-way tradeoff when $b = 2$, 4, and 8, respectively. The first column shows the range of U values that produces the same g-counter threshold T in the second column. The third column shows the minimum number of bits each g-counter must have. The fourth column shows the number of uniform hashes that need to be performed in order to produce one geometric hash output.

For example, according to Table II, when $b = 2$, $U \in [381, 763]$ will all have the same g-counter threshold $T = 7$. That means even if $U = 763$, a flow of smaller thread $n_f = 381$ will still have a 99% chance to pass the filter. When U is set sufficiently high, this is not a serious problem in practice since the vast majority of flows have much smaller spread values (as we have observed from the CAIDA traffic and our campus network traffic), which means a majority of small flows will be blocked as our experiments based on real traffic traces will show. According to Table III, when $b = 4$, if $U = 763$, $T = 16$; if $U = 381$, $T = 14$. They have different threshold values at the cost of higher geometric counter size, 5 or 4 bits v.s. 3 bits when $b = 2$. According to Table IV, when $b = 8$, if $U = 763$, $T = 36$; if $U = 381$, $T = 31$. They are further apart at the cost of higher g-counter size, 6 or 5 bits, and more hash bits, 108 or 93 v.s. 32 or 28 hash bits when $b = 4$.

TABLE II
RESOURCE NEEDED FOR A G-COUNTER WHEN $b = 2$

U	T	No. of bits	No. of hash bits
9-20	1	1	1
21-44	2	2	2
45-92	3	2	3
93-188	4	3	4
189-380	5	3	5
381-763	6	3	6
764-1529	7	3	7
1530-3062	8	4	8
3063-6128	9	4	9
6129-12259	10	4	10
12260-24522	11	4	11

TABLE III
RESOURCE NEEDED FOR A G-COUNTER WHEN $b = 4$

U	T	No. of bits	No. of hashes bits
5-7	1	1	2
8-10	2	2	4
11-15	3	2	6
16-22	4	3	8
23-30	5	3	10
31-41	6	3	12
...	...	...	...
104-138	10	4	20
139-186	11	4	22
187-249	12	4	24
250-333	13	4	26
334-445	14	4	28
446-594	15	4	30
595-793	16	5	32
...	...	...	...
4473-5964	23	5	46
5965-7953	24	5	48
7954-10605	25	5	50
10606-14142	26	5	52
14143-18857	27	5	56

According to Table IV, when $b = 8$, if $U = 763$, $T = 36$; if $U = 381$, $T = 31$. They are further apart at the cost of higher g-counter size, 6 or 5 bits, and more hash bits, 108 or 93 v.s. 32 or 28 hash bits when $b = 4$.

We now present the following theorem to show the number of uniform hash operations needed to process a packet. Due to space limitation, we do not put proofs here. Readers can find the proofs in supplemental materials or the appendix part of this paper' full version [62].

Theorem 4: In our geometric min filter, if the length of the output of the hash operation we use is l_{hash}, then the number of hash operations needed for processing one packet is

$$n_{hash} = \frac{d(\lceil T\log_2 b\rceil + \lceil \log_2 l\rceil)}{l_{hash}} + 1 \quad (16)$$

with T the geometric threshold, b the base of g-counter, l the length of each g-counter array and d the number of arrays in the filter.

Corollary 2: The time complexity of our filter is

$$O(2(d+1)T_M + (\frac{d(\lceil T\log_2 b\rceil + \lceil \log_2 l\rceil)}{l_{hash}} + 1)T_H). \quad (17)$$

TABLE IV
RESOURCE NEEDED FOR A G-COUNTER WHEN $b = 8$

U	T	No. of bits	No. of hash bits
3-4	1	1	3
5-5	2	2	6
6-6	3	2	9
7-8	4	3	12
9-10	5	3	15
11-12	6	3	18
13-14	7	3	21
15-16	8	4	24
...	...	...	...
326-372	30	5	90
373-426	31	5	93
427-487	32	6	96
488-558	33	6	99
559-638	34	6	102
639-729	35	6	105
730-834	36	6	108
...	...	...	...
7091-8103	53	7	159
8104-9261	54	7	162
9262-10584	55	7	165
10585-12097	56	7	168
12098-13826	57	7	171
13827-15801	58	7	174
15802-18059	59	7	177

Here, T is the geometric threshold that can be obtained from Theorem 1; l is the length of a filter array that can be obtained from Theorem 3; b is the base of g-counters in the filter; d is the number of arrays in the filter; l_{hash} is the length of the output of a hash operation; $O(T_M)$ and $O(T_H)$ are the time complexity of a memory access and a hash operation, respectively.

We also give the hash bits needed by vSkt(HLL) here. According to the paper [29], with the suggested setting, vSkt(HLL) requires $32 + \lceil log_2\ l_{vskt} \rceil$ hash bits for recording an incoming packet, where l_{vskt} is the number of registers in vSkt(HLL).

D. Network-Wide Measurement

We are not only considering TCP flows whose packets generally follow the same paths. In our generalized model, the packets of a flow may follow different paths. For example, consider a network with two gateways to the Internet. Suppose we monitor all outbound packets that carry URL requests. If we use URL (in the application header) as flow identifier, all URL requests for the same web page form a flow. The packets of any flow may pass two gateways. If we use source address as element identifier, the spread of a flow is the number of different hosts that access the same web page. If we set the threshold for super spreaders to be U, we will need to combine the measurements from both gateways in order to identify super spreaders. This is an example of network-wide measurement.

Let q be the number of network devices that jointly monitor super spreaders. One observation is that, for any super spreader f, at least one device will receive at least $U' = \frac{U}{q}$ of its elements. Based on this observation, we set U' as the spread threshold at each device and collect all potential super spreaders in its hash table. After all devices send their filters, hash tables and CDS to the server. The server will merge the filters as follows: Denote the filter from the kth device as F^k and denote the jth g-counter of the i array in F^k as $F_i^k[j], 0 \le k < q, 0 \le i < d, 0 \le j < l$. Note that we require all filters to have the same dimensions of d and l. Denote the resulting filter after merging as F^*. The merge operation is

$$F_i^*[j] = \max\{F_i^k[j], 0 \le k < q, 0 \le i < d, 0 \le j < l\}. \quad (18)$$

In order to support the merging operation, we have to keep updating the filter after a flow passes it. See the revised algorithm in Alg. 3 below, where the hash table and the CDS at the kth device are denoted as HT^k and CDS^k, respectively. Even after a flow is inserted into HT^k, we have to record its elements in F^k.

Algorithm 3 Geometric-Min Filter at Device k

Input: filter F^k, b, U, α, q
Action: hash table HT^k

$T = \lfloor \log_{\frac{b-1}{b}}(1 - (1 - \alpha^{\frac{1}{d}})^{\frac{1}{U/q}}) \rfloor$
for each arrival packet $\langle f, e \rangle$ **do**
  $v(f) = INT\_MAX$
  for $i = 0..d-1$ **do**
    $F_i^k[H_i(f)] = \max\{F_i^k[H_i(f)], G_{i,b}(f, e)\}$
    $v(f) = \min\{F_i^k[H_i(f)], v(f)\}$
  end for
  if $v(f) \ge T$ **then**
    Insert f to HT^k
  end if
end for
return HT^k

After we compute $F*$, we iterate through all flow identifiers f in HT^k, $0 \le k < q$, and insert those with $v^*(f) \ge T^*$ into a new table HT^*, where

$$v^*(f) = \min\{F_i^*[H_i(f)], 0 \le i < d\}$$
$$T^* = \lfloor \log_{\frac{b-1}{b}}(1 - (1 - (1 - \gamma)^{\frac{1}{d}})^{\frac{1}{U}}) \rfloor. \quad (19)$$

We claim that CDS^k, $0 \le k < q$ should also support merging operation if users want to use it to get an estimation of super spreaders here. Actually, vSkt(HLL) [29] can support that. Let the merged CDS be CDS*. For each flow in HT^*, we estimate the flow spread from CDS*.

VI. EVALUATION

We evaluate the performance of geometric-min filter using both software and hardware implementation.

Software Implementation: All the solutions are implemented in java on a desktop with Intel Core i7-8700 3. 2GHz CPU and 16GB memory.

Hardware Implementation: We implement the geometric-min filter on a XILINX Nexys4 A7-100T development board, with 15850 logic slices, 4860Kbits Block RAM, and a clock rate of 100MHz. More specifically, we implement the filter part on the FPGA board using Vivado [63]. The block ram is used for the filter array. The recording operation of each array is implemented as a module that takes $\langle f, e \rangle$ as the input, updates the array and outputs the register value. We use FIFO modules to connect d modules for d arrays. Another module is placed at the end which calculates the minimum register value and outputs the flow label if it exceeds the threshold.

The hash table is supposed to be implemented on the software platform since implementing a hash table on FPGA is a waste of resources. The checking of hash table before updating the filter is omitted here. As we have discussed in Section V-A, this operation will not affect the filtering performance. We also implement vSkt(HLL) [29], the detail can be found in the original paper.

Since the only difference between hardware and software implementation will be the throughput, when we do other comparisons, we only focus on the software platform. The codes can be found at [64].

In the evaluation, we first evaluate the performance of our filter under different parameter settings and then use the best setting to compare the performance with the state of art including SpreadSketch [38] and AROMA [35].

A. Traffic Summary

We download traffic traces from CAIDA15, CAIDA18 and CAIDA19 on CAIDA [39] for experiments. For each data set (CAIDA15, CAIDA18 or CAIDA19), we download 10 traffic traces of 1-minute long. The result for each data set is the average result of the 10 traffic traces. We use destination IP address as flow identifier and source IP address as element identifier. Two packets are identical only when both the source and destination IP addresses are the same. Under this definition, each trace from CAIDA15 contains around 18M packets, 120K flows and 430K distinct packets; each trace from CAIDA18 contains around 27M packets, 270K flows and 900K distinct packets; each trace from CAIDA19 contains around 36M packets, 750K flows and 1800K distinct packets. We also create 10 traffic traces to form another data set called CAIDA15+ by adding a large flow with spread of 2M into each trace in CAIDA15.

B. Performance Metrics

We first define four terms to facilitate the understanding of the performance metrics.

True Positive (TP): The number of super spreaders passing the filter.

False Positive (FP): The number of non super spreaders passing the filter.

True Negative (TN): The number of non super spreaders not passing the filter.

False Negative (FN): The number of super spreaders not passing the filter.

We use the following metrics to evaluate the performance of our whole solution.

Throughput: The number of packets the system can process in one second during online encoding. The unit is packets per second which is denoted as pps.

False Negative Ratio(FNR): $\frac{FN}{FN+TP}$. It means the probability of a super spreader not passing the filter.

False Positive Ratio(FPR): $\frac{FP}{FP+TN}$. It represents the probability of a non super spreader reported as a super spreader.

Precision: $\frac{TP}{TP+FP}$. It is the probability that a reported super spreader is a real super spreader.

Recall: $\frac{TP}{FN+TP}$. It is the probability of a real super spreader being reported.

F1 Score: $\frac{2}{recall^{-1}+precision^{-1}}$. It is the harmonic mean of precision and recall.

Since the parameter setting of our filter is related to FNR, we first use FNR and FPR as the metrics for the experiments over different parameter settings for our filter. After that, when comparing our solution with existing solutions, we use Precision, Recall and F1 Score as the metrics because they are the metrics adopted by the authors of existing solutions.

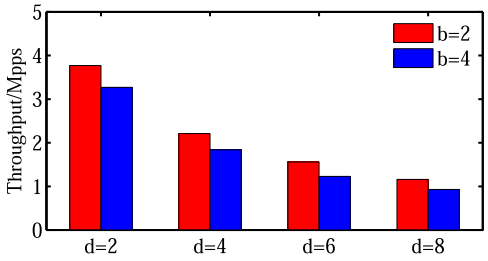


Fig. 2. Throughput w.r.t. d and b. Throughput will decrease when b and d increase.

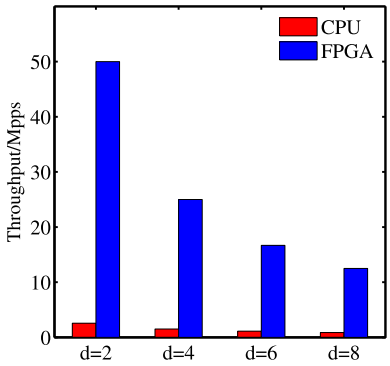


Fig. 3. Throughput comparison between FPGA implementation and CPU implementation. Throughput will decrease as d increases. FPGA implementation is 14.5-19.56 times faster than CPU implementation.

C. Throughput Comparison Under Different Parameter Settings

We compare the throughput of our geometric-min filter under different parameter settings on hardware/software platforms.

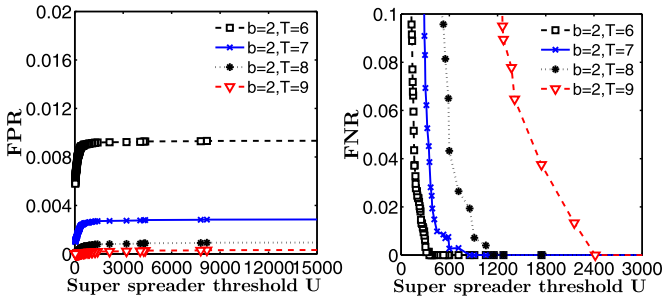
Fig. 2 shows, on CPU implementation, the throughput of our solution decreases as d increasing. This is predictable because we have to encode the filter d times for a packet of flow f before it passes the filter. Besides, with geometric counters of a larger base b in the filter, we will have a smaller throughput. The reason is that with a larger b, we need to compute more hash functions to get the required geometric hash values as we describe in Section IV.

Since on FPGA we can use FIFO module to pipeline the processing operations, we obtain a throughput that equals to $\frac{1}{d}$ packet per clock cycle while b will not affect the throughput. The clock rate of the board we use is 100MHz, so the throughput will be 50, 25, 16.67, 12.5 Mpps, which is consistent with our experiment result. As Fig. 3 shows, it is much faster than the Software Implementation on CPU.

D. FNR/FPR Comparison Under Different Parameter Settings

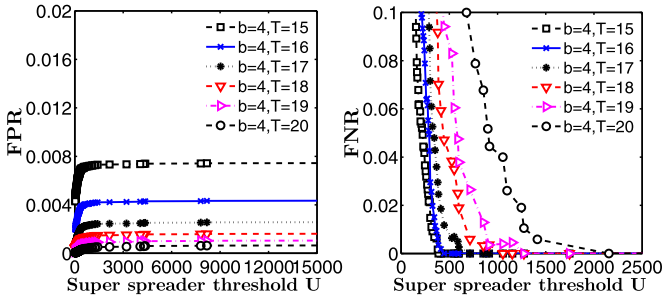
In this section, we evaluate the impact of parameter settings for our geometric-min filter over FNR/FPR.

1) Comparison Under Different T Values: First, we set $d = 4, l = 5120, b = 2, 4$ and change T to compare the influences of T over FPR/FNR under different counter base b.



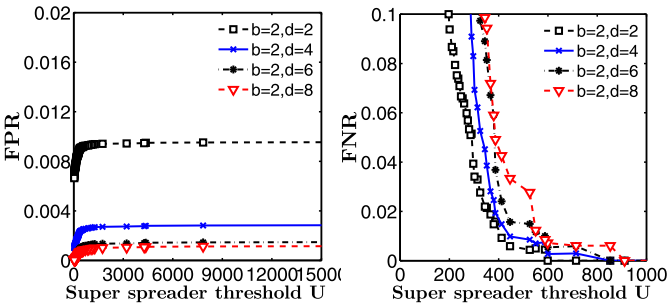
(a) Average FPR (b) Average FNR

Fig. 4. FNR/FPR w.r.t. T when $b = 2$.



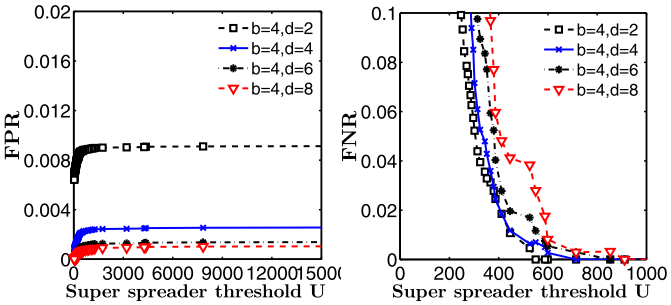
(a) Average FPR (b) Average FNR

Fig. 5. FNR/FPR w.r.t. T when $b = 4$.



(a) Average FPR (b) Average FNR

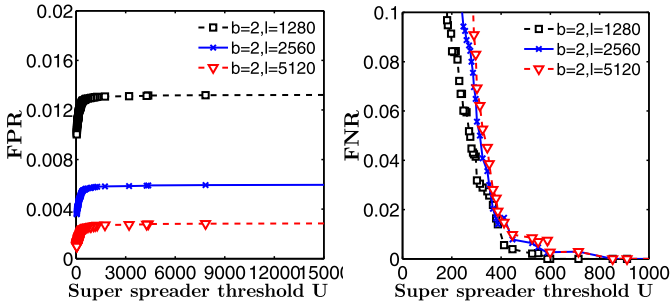
Fig. 6. FNR/FPR w.r.t. d when $b = 2$.



(a) Average FPR (b) Average FNR

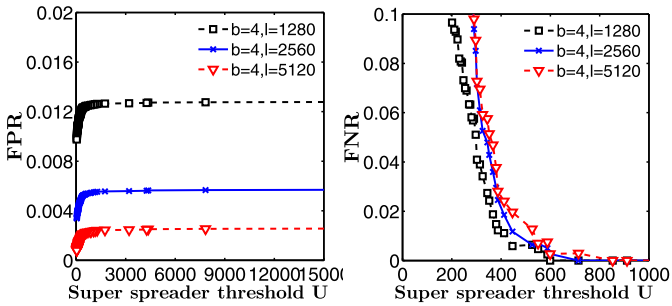
Fig. 7. FNR/FPR w.r.t. d when $b = 4$.

From Fig. 4 and 5, we observe that the average FNR of our solution will decrease when U becomes larger and with a larger T, we can get a lower FNR for a same U. This is consistent to equation (11), if we want to guarantee a same FNR, we need to choose a larger T for a larger U. For FPR, we can see from Fig.4 and 5 that a larger T will always bring a smaller FPR. This means we have to choose a larger T as long as we can guarantee the FNR just as what we get in equation (11). This can be confirmed by comparing the lines in Fig. 4 and 5 with the values in Table II and Table III, which shows that our computed best T actually guarantee the FNR bound $\alpha = 0.99$ while keeping a smallest possible FPR.



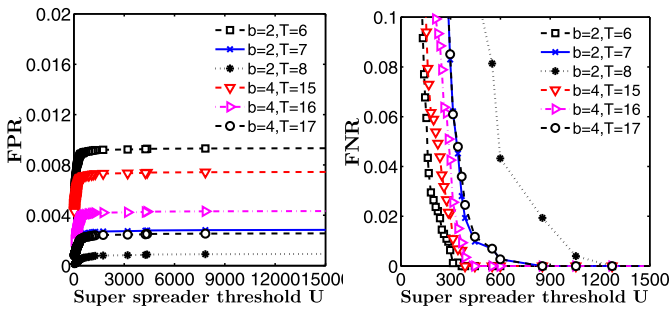
(a) Average FPR (b) Average FNR

Fig. 8. FNR/FPR w.r.t. l when $b = 2$.



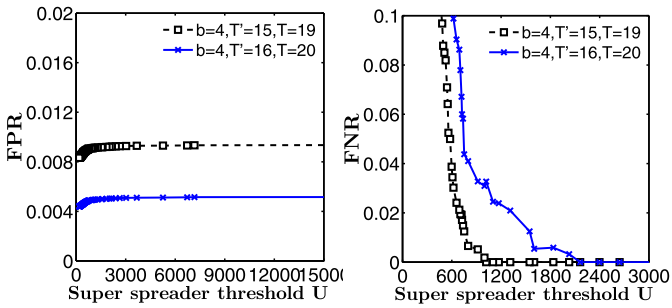
(a) Average FPR (b) Average FNR

Fig. 9. FNR/FPR w.r.t. l when $b = 4$.



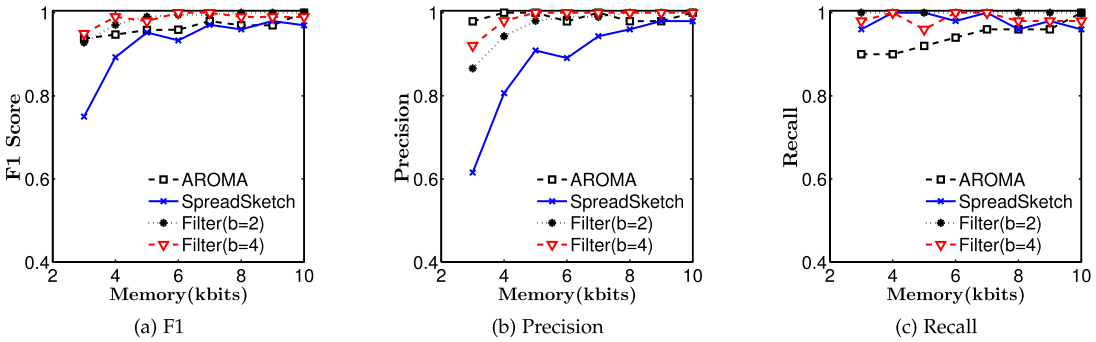
(a) Average FPR (b) Average FNR

Fig. 10. FNR/FPR w.r.t. *b and T*.



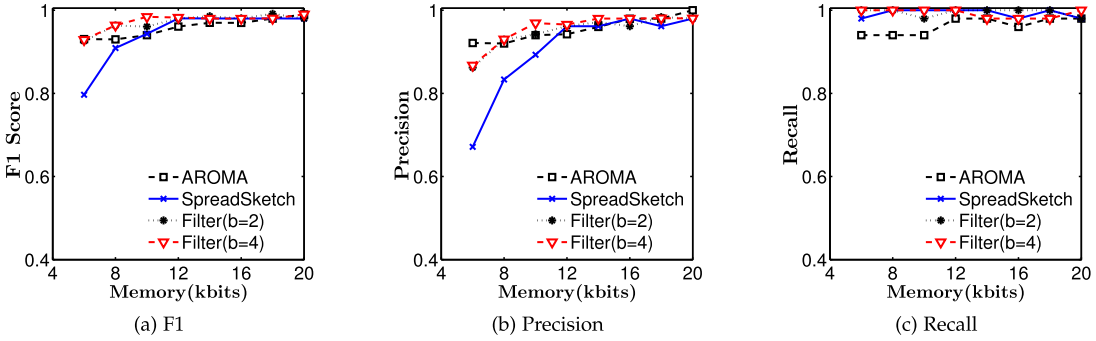
(a) Average FPR (b) Average FNR

Fig. 11. FNR/FPR for network-wide measurement.



(a) F1 (b) Precision (c) Recall

Fig. 12. Comparisons with different memory allocations over CAIDA15 data sets.



(a) F1 (b) Precision (c) Recall

Fig. 13. Comparisons with different memory allocations over CAIDA18 data sets.

2) Comparison Under Different d Values: To compare the performance of the filter on FPR/FNR between d, we keep $l = 5120, b = 2, T = 7$ or $l = 5120, b = 4, T = 17$ and change $d = 2, 4, 6, 8$. Fig. 6 and 7 show that with a larger d, we can have a smaller FPR which means the filter can filter out more non super spreaders. For $b = 2$, changing d from 2 to 8 can reduce the FPR from 0.01 to 0.0012 and for $b = 2$, changing d from 2 to 8 can reduce the FPR from 0.009 to 0.001. The reason is that with a large d, the probability of flow passing the filter will be smaller. Since we can choose suitable T for different U and d to guarantee the FNR bound, setting a larger d is always better.

3) Comparison Under Different l Values: We keep $d = 4, b = 2, T = 7$ and $d = 4, b = 4, T = 17$ and change $l = 1280, 2560, 5120$ to learn the impact of l on FPR/FNR of the filter. According to Fig. 8 and 9, we know that with a smaller l, we will obtain a much larger FPR and a slight smaller FNR for a certain U. We claim that we should choose l as large as possible since that the threshold calculated from equation(11) can always guarantee the FPR bound for any l and a larger l will always give a smaller FPR. The reason that a larger l will give a smaller FPR is that with a larger l, the probability of a small flow sharing a same geometric counter with a large flow will be lower and thus the probability of small flows passing the filter will be lower. In this situation, a larger l can let fewer non super spreaders be misidentified as super spreaders. However, a larger l will take larger memory.

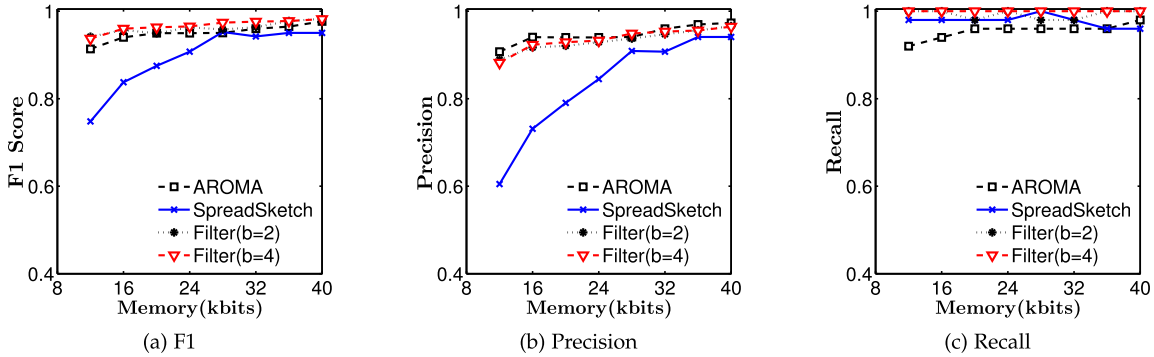
4) Comparison Under Different b Values: As we discuss in Section V-C, using geometric counter with $b > 2$ have advantages in choosing suitable T for different user-defined spread threshold U. The optimal T can be calculated from equation (11). Some results are already shown in Table II and III when we want to guarantee the bound of FNR as $\alpha = 0.99$. From Table II we know that when $U \in [381, 763], b = 2$, the best choice of T are all 7. From Fig. 10, we know that when $b = 2, T = 6$, the average FNR when $U \leq 381$ is actually lower than $\alpha = 0.99$ while keeping an average FPR close to 0.06. From Table III, we know that when $b = 4$ and $U = 500, 600$, we should choose $T = 15, 16$ respectively. According to Fig. 10, the average FNR all meet the bound $\alpha = 0.99$. Meanwhile, the average FPR is at most around $0.007, 0.004$ which are significantly less than choosing $T = 6$ when $b = 2$. With a similar FNR, a smaller FPR can certainly reduce the memory for storing super spreader labels in the hash table. Besides, from Fig. 10b, we can see that the influence of FNR by adding 1 to T when $b = 2$ is much larger than that when $b = 4$. This implies that a larger b can divide the range of U more delicately for user to choose suitable T just as what we can see from Table II, Table III and Table IV.

E. Network-Wide Measurement

In this section, we measure the FNR/FPR performance of geometric-min filter for network-wide super spreader identification as we describe in Section V-D. The number of devices in this experiment is 3. We tried two settings that $T' = 15, T = 19$ and $T' = 16, T = 20$ where T' is the threshold in each network device and T is the threshold for the merged filter. Actually these two settings is for $U = 1500$ and $U = 2100$. Fig. 11 shows that our method can still guarantee the FNR as $\alpha = 0.99$ and keep a FPR up to $0.009, 0.005$ respectively which are all acceptable.

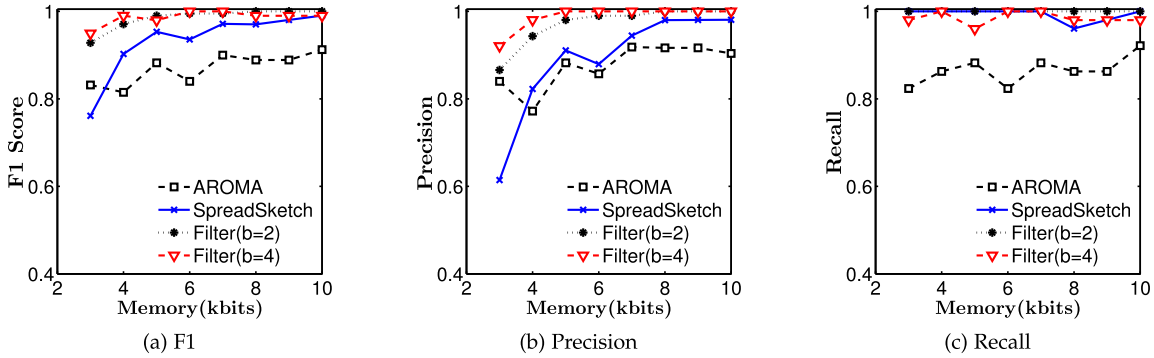
F. Comparison With Existing Solutions

1) Comparisons Under Different Traffic Traces: We compare our algorithms with the state of the art, i.e., SpreadSketch [38] and AROMA [35] under different data sets from CAIDA [39]. By the years when the data sets are generated, they are denoted as CAIDA15, CAIDA18 and CAIDA19, respectively. We use 10 different traces of 1 minute long in



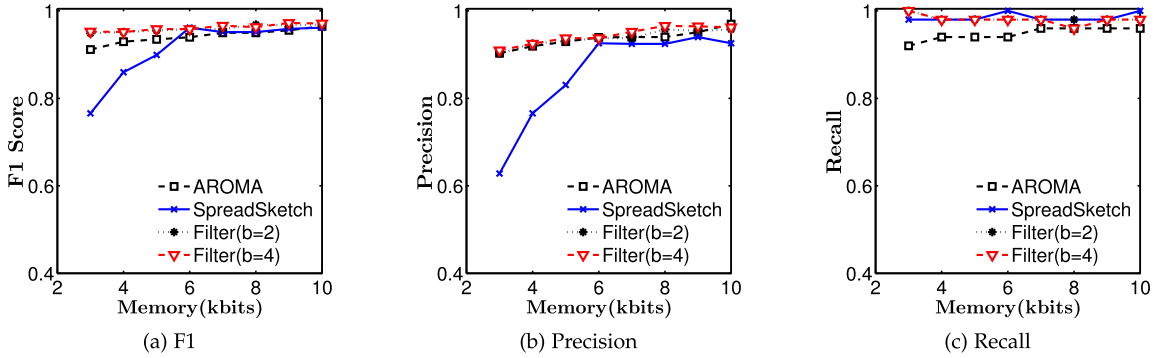
(a) F1 (b) Precision (c) Recall

Fig. 14. Comparisons with different memory allocations over CAIDA19 data sets.



(a) F1 (b) Precision (c) Recall

Fig. 15. Comparisons with different memory allocations over CAIDA15+ data sets.



(a) F1 (b) Precision (c) Recall

Fig. 16. Comparisons with different memory allocations for network-wide measurement.

each year for experiments and show the average results. For each trace, we tune the value of threshold U that guarantees the number of super spreaders is around 50. For our solution, we use $\alpha = 0.95$ to calculate the filter threshold T according to Theorem 1 and then calculate l according to Theorem 3 using $\gamma = 0.95$ and $L = lower\ bound\ of\ 1\%\ top\ flows$. After that, we assign the rest memory for vSkt(HLL) [29]. We vary the memory allocation for all solutions and apply them on different traces. Since the number of packets and flows are different for CAIDA15, CAIDA18, CAIDA19, the range of memory allocation are also different. Traces in CAIDA19 contains the most packets and flows, the memory range is widest, i.e., [12,40] Mb; traces in CAIDA15 contains the least packets and flows, the memory range is narrowest, i.e., [3,10] Mb.

The results in Figs. 12, 13 and 14 show that for all data sets, all solutions will have a larger F1 score for a larger memory allocation. All solutions can have a F1 score larger than 0.9 when memory is large enough (5Mbits for CAIDA15, 8Mbits for CAIDA18 and 24Mbits for CAIDA19). Our solution will have a highest F1 score when memory is larger than a certain value (4Mbits for CAIDA15, 8Mbits for CAIDA18 and 10Mbits for CAIDA19) while SpreadSketch always have a lowest F1 score. For small memory, AROMA and our solution have close F1 score. For all traces, our solution has highest recalls while AROMA has the lowest recalls. As for precision, our solution is similar to AROMA while SpreadSketch has low precisions when memory is small.

We also compare the above algorithms on CAIDA15+ data sets to show the limitation of AROMA. We create the data sets by adding a fake flow with spread of 2M into each traces in CAIDA15. We run all three algorithms on those traces. Comparing Fig. 15 with Fig. 12, the F1 score of AROMA drops significantly for these special traces while the F1 scores of SpreadSketch and our solution have little changes, which is predictable. The extremely large flow will occupy a high percentage of buckets in AROMA. Therefore, the estimation for other super spreaders will be inaccurate. In the meanwhile, a flow will only occupy one or several units (registers) for

SpreadSketch and our solution. This means even an extremely large flow will have little influence on the performance.

2) Comparisons for Network-Wide Measurement: We compare our algorithms with SpreadSketch [38] and AROMA [35] for network-wide measurement. The number of devices in this experiment is 3. We combined 3 1-minute traffic traces and randomly assign the packets in the combined trace to three devices to simulate the situation. The results we show are for traces in CAIDA15, the results for CAIDA18 and CAIDA19 are similar to the first one and are thus omitted here. The parameter setting is similar to Section VI-F1. Fig. 16 shows that all solutions will have a F1 score larger than 0.9 when memory is larger than 5Mbits. When memory is larger than 3Mbits, our solution will have the highest F1 score. For precision, our solution and AROMA are better than SpreadSketch, while for recall, our solution and SpreadSketch are better than AROMA.

VII. Conclusion

In this paper, we propose a geometric-min filter solution for the problem of super spreader identification based on two technical innovations, generalized geometric hash and generalized geometric counter. The solution has a better super spreader identification performance when comparing with the state-of-the-art work SpreadSketch [38] and AROMA [35] under different memory allocations and traffic traces. We implement the solution in both hardware and software.

References

[1] S. Chen and Y. Tang, "Slowing down internet worms," in *Proc. 24th Int. Conf. Distrib. Comput. Syst.*, Mar. 2004, pp. 312–319.

[2] J. Gong *et al.*, "HeavyKeeper: An accurate algorithm for finding top-*k* elephant flows," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*. Boston, MA, USA: USENIX Association, 2018, pp. 909–921. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/gong

[3] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (IEEE INFOCOM)*, Apr. 2016, pp. 1–9.

[4] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. 10th Int. Conf. Database Theory (ICDT)*, Jan. 2005, pp. 398–412.

[5] G. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. VLDB*, Aug. 2002, pp. 346–357.

[6] E. Demaine, A. Lopez-Ortiz, and J. Munro, "Frequency estimation of internet packet streams with limited space," in *Proc. 10th Annu. Eur. Symp. Algorithms (ESA)*, Sep. 2002, pp. 348–360.

[7] 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. ACM SIGCOMM Conf.*, Aug. 2016, pp. 101–114.

[8] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 561–575.

[9] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets," in *Proc. Conf. Internet Meas.*, Oct. 2004, pp. 115–120.

[10] N. Kamiyama and T. Mori, "Simple and accurate identification of high-rate flows by packet sampling," in *Proc. IEEE INFOCOM*, Apr. 2006, pp. 1–13.

[11] Y. Yao, S. Xiong, J. Liao, M. Berry, H. Qi, and Q. Cao, "Identifying frequent flows in large datasets through probabilistic bloom filters," in *Proc. 23rd Int. Symp. Qual. Service*, Jun. 2015, pp. 279–288.

[12] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational IP networks: Methodology and experience," *IEEE/ACM Trans. Netw.*, vol. 9, no. 3, pp. 265–279, Jun. 2001.

[13] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerg. Netw. EXperiments Technol. (CoNEXT)*, 2011, pp. 1–12.

[14] J. Rasley *et al.*, "Planck: Millisecond-scale monitoring and control for commodity networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 407–418, 2015.

[15] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, Jun. 2014, pp. 228–237.

[16] J. A. Copeland, "Flow-based detection of network intrusions," U.S. Patent 7 185 368, Feb. 27, 2007.

[17] G. A. Ajaeiya, N. Adalian, I. H. Elhajj, A. Kayssi, and A. Chehab, "Flow-based intrusion detection system for SDN," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2017, pp. 787–793.

[18] R. Chandra, "Network traffic monitoring for search popularity analysis," U.S. Patent 7 594 011, Sep. 22, 2009.

[19] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic DDos defense," in *Proc. 24th USENIX Secur. Symp. (USENIX Secur.)*, 2015, pp. 817–832.

[20] M. Wischenbart *et al.*, "User profile integration made easy: Model-driven extraction and transformation of social network schemas," in *Proc. 21st Int. Conf. Companion World Wide Web (WWW Companion)*, 2012, pp. 939–948.

[21] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser, "Processing a trillion cells per mouse click," 2012, *arXiv:1208.0225*. [Online]. Available: http://arxiv.org/abs/1208.0225

[22] S. Melnik *et al.*, "Dremel: Interactive analysis of web-scale datasets," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 330–339, Sep. 2010.

[23] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proc. 16th Int. Conf. Extending Database Technol. (EDBT)*, 2013, pp. 683–692.

[24] P. Lieven and B. Scheuermann, "High-speed per-flow traffic measurement with probabilistic multiplicity counting," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.

[25] M. Yoon, T. Li, S. Chen, and J.-K. Peir, "Fit a spread estimator in small memory," in *Proc. 28th Conf. Comput. Commun. (IEEE INFOCOM)*, Apr. 2009, pp. 504–512.

[26] Q. Xiao, S. Chen, M. Chen, and Y. Ling, "Hyper-compact virtual estimators for big network data based on register sharing," in *Proc. Int. Conf. Meas. Modeling Comput. Syst. (ACM SIGMETRICS)*, Jun. 2015, pp. 417–428.

[27] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen, "Estimating the persistent spreads in high-speed networks," in *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, Oct. 2014, pp. 131–142.

[28] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high-speed links," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 925–937, Oct. 2006.

[29] Y. Zhou, Y. Zhang, C. Ma, S. Chen, and O. O. Odegbile, "Generalized sketch families for network traffic measurement," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 3, pp. 1–34, Dec. 2019, doi: 10.1145/3366699.

[30] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proc. Symp. Netw. Syst. Design Implement. (USENIX)*, 2013, pp. 29–42.

[31] H. Wang, C. Ma, O. O. Odegbile, S. Chen, and J.-K. Peir, "Randomized error removal for online spread estimation in data streaming," *Proc. VLDB Endowment*, vol. 14, no. 6, pp. 1040–1052, Feb. 2021.

[32] C. Ma, H. Wang, O. Odegbile, and S. Chen, "Virtual filter for non-duplicate sampling," in *Proc. IEEE Int. Conf. Netw. Protocols (ICNP)*, Nov. 2021, pp. 1–11.

[33] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum, "New streaming algorithms for fast detection of superspreaders," in *Proc. NDSS*, 2005, pp. 1–18.

[34] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani, "Streaming algorithms for robust, real-time detection of DDoS attacks," in *Proc. 27th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2007, p. 4.

[35] R. B. Basat, X. Chen, G. Einziger, S. L. Feibish, D. Raz, and M. Yu, "Routing oblivious measurement analytics," in *Proc. IFIP Netw. Conf. (Netw.)*, 2020, pp. 449–457.

[36] G. Cormode and S. Muthukrishnan, "Space efficient mining of multigraph streams," in *Proc. ACM PODS*, 2005, pp. 271–282.

[37] Y. Liu, W. Chen, and Y. Guan, "Identifying high-cardinality hosts from network-wide traffic measurements," *IEEE Trans. Dependable Secure Comput.*, vol. 13, no. 5, pp. 547–558, Sep. 2016.

[38] L. Tang, Q. Huang, and P. P. C. Lee, "SpreadSketch: Toward invertible and network-wide detection of superspreaders," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Jul. 2020, pp. 1608–1617.

[39] *CAIDA*. Accessed: 2021. [Online]. Available: https://www.caida.org/data

[40] Q. Zhao, J. Xu, and Z. Liu, "Design of a novel statistics counter architecture with optimal space and time efficiency," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, pp. 323–334, 2006.

[41] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: A novel counter architecture for per-flow measurement," in *Proc. ACM SIGMETRICS*, Jun. 2008, pp. 121–132.

[42] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better NetFlow for data centers," in *Proc. USENIX NSDI*, 2016, pp. 311–324.

[43] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *Proc. LATIN*, 2004, pp. 58–75.

[44] Y. Lu and B. Prabhakar, "Robust counting via counter braids: An error-resilient network measurement architecture," in *Proc. 28th Conf. Comput. Commun. (IEEE INFOCOM)*, Apr. 2009, pp. 522–530.

[45] Y. Zhou, Y. Zhou, S. Chen, and Y. Zhang, "Highly compact virtual active counters for per-flow traffic measurement," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 1–9.

[46] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *IEEE/ACM Trans. Netw.*, vol. 20, no. 5, pp. 1622–1634, Oct. 2012.

[47] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. 1st ACM SIGCOMM Workshop Internet Meas. (IMW)*, 2001, pp. 323–336.

[48] M. Chen and S. Chen, "Counter tree: A scalable counter architecture for per-flow traffic measurement," in *Proc. IEEE 23rd Int. Conf. Netw. Protocols (ICNP)*, Nov. 2015, pp. 1249–1262.

[49] Cisco. (2019). *Cisco IOS NetFlow*. [Online]. Available: http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html

[50] C. Ma, H. Wang, O. Odegbile, and S. Chen, "Noise measurement and removal for data streaming algorithms with network applications," in *Proc. IFIP Netw. Conf. (IFIP Netw.)*, Jun. 2021, pp. 1–9.

[51] Y. Zhou, Y. Zhou, M. Chen, and S. Chen, "Persistent spread measurement for big network data based on register intersection," in *Proc. ACM SIGMETRICS*, Jun. 2017, pp. 1–29.

[52] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 208–229, Jun. 1990.

[53] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for database applications," *J. Comput. Syst. Sci.*, vol. 31, pp. 182–209, Sep. 1985.

[54] M. Durand and P. Flajolet, "LogLog counting of large cardinalities," in *Proc. Eur. Symposia Algorithms (ESA)*, 2003, pp. 605–617.

[55] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. AOFA*, 2007, pp. 127–146.

[56] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 127–140.

[57] Q. Huang, P. P. C. Lee, and Y. Bao, "SketchLearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proc. SIGCOMM*, Aug. 2018, pp. 576–590.

[58] Z. Liu *et al.*, "NitroSketch: Robust and general sketch-based monitoring in software switches," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 334–350.

[59] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.

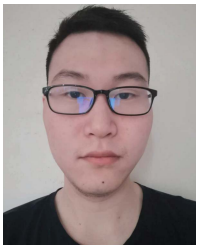
[60] G. Einziger and R. Friedman, "Counting with TinyTable: Every bit counts!" *IEEE Access*, vol. 7, pp. 166292–166309, 2019.

[61] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proc. ACM SIGCOMM Conf. Internet Meas. (IMC)*, 2003, pp. 153–166.

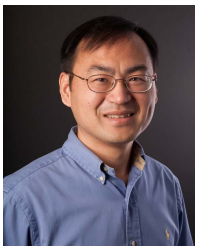
[62] *Full Version*. Accessed: 2021. [Online]. Available: https://www.dropbox.com/s/lokjzo5w8dea3s8/paper.pdf?dl=0

[63] *VIVADO*. Accessed: 2021. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[64] *Source Code for Geometric-Min Filter*. Accessed: 2021. [Online]. Available: https://github.com/mcynever/GeometricMinFilter



Chaoyi Ma (Graduate Student Member, IEEE) received the B.S. degree in computer information security from the University of Science and Technology of China in 2018. He is currently pursuing the Ph.D. degree in computer and information science and engineering with the University of Florida, under the supervision of Prof. S. Chen. His research interests include big data, network traffic measurement, computer network security, and data privacy in machine learning.



Shigang Chen (Fellow, IEEE) received the B.S. degree in computer science from the University of Science and Technology of China in 1993, and the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana–Champaign in 1996 and 1999, respectively. After graduation, he had worked with Cisco Systems for three years before joining the University of Florida in 2002. He is currently a Professor with the Department of Computer and Information Science and Engineering, University of Florida. He has published over 200 peer-reviewed journal articles/conference papers. He holds 13 U.S. patents and many of them were used in software products. His research interests include big data, the Internet of Things, cybersecurity, RFID technologies, and intelligent cyber-transportation systems. He received the NSF CAREER Award and several best paper awards. He served as an Associate Editor for IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE/ACM TRANSACTIONS ON NETWORKING, and a number of other journals. He served in various chair positions or as a committee member for numerous conferences. He is an ACM Distinguished Scientist.

Youlin Zhang received the B.S. degree in electronic information engineering from the University of Science and Technology of China, Hefei, China, in 2014, and the Ph.D. degree in computer and information science and engineering from the University of Florida, Gainesville, FL, USA, in 2019. His research interests include big network data and the Internet of Things.

Qingjun Xiao (Member, IEEE) received the B.Sc. degree from the Department of Computer Science, Nanjing University of Posts and Telecommunications, China, in 2003, the M.Sc. degree from the Department of Computer Science, Shanghai Jiao Tong University, China, in 2007, and the Ph.D. degree from the Department of Computer Science, The Hong Kong Polytechnic University, in 2011. After receiving the Ph.D. degree, he joined the Department of Computer Science, Georgia State University and the University of Florida, and held a postdoctoral position for three years combined. He is currently an Associate Professor with Southeast University, China. His research interests include network traffic measurement, data stream processing, and cybersecurity. He is a member of ACM and China Computer Federation (CCF).

Olufemi O. Odegbile received the B.S. degree in mathematics from the University of Ibadan, Nigeria, the master's degree in computer science from Boston University, USA, and the Ph.D. degree in computer science from the University of Florida. He is currently an Assistant Professor with the Department of Computer Science, Clark University, Worcester, USA. His research interests include computer networks, network security, network traffic measurement, and RFID technology.