# PS-Sketch: Fast and Accurate Detection of Persistent-Spreaders in High-Speed Networks

Zhaohui Wang, Ye Tian, *Member, IEEE,* Yiwen Wu, Wei Chen,
Xinyu Zhang, and Xinming Zhang, *Senior Member, IEEE*

***Abstract*—Large spread and high persistence are widely observed in malicious activities such as botnets and DDoS attacks in high-speed networks, while how to identify persistent-spreaders is still a challenging issue. In this work, we present *PS-Sketch*, a system for estimating persistent-spreads of network flows and detecting persistent-spreaders in network data streams. PS-Sketch is based on our definitions of persistence and persistent-spread, which overcome the limitations of the conventional definitions by better capturing network flows' behaviors, and being difficult for attackers to bypass. Within a switch's pipeline, PS-Sketch processes packets with two adjacent sketch data structures, namely the P-sketch and the S-sketch. In the P-sketch, we employ low-pass filter (LPF) to trace an element's persistence that incorporates occurrences in its entire history, and in the S-sketch, we extend the HyperLogLog (HLL) algorithm, and integrate an element's persistence to the spread estimation of the flow that the element belongs to, for estimating the flow's persistent-spread. We present theoretical analysis on the error bound of PS-Sketch. Trace-driven evaluation shows that PS-Sketch achieves high accuracy in estimating network flows' persistent-spreads, and outperforms the existing solutions in detecting persistent-spreaders. We further prototype PS-Sketch in P4 and show that the system is deployable on commodity hardware switches.**

***Index Terms*—Persistence, persistent-spreader, sketch, programmable switch**

## I. Introduction

In large-scale network attacks such as DDoS attacks [1], network scanning [2], worm propagation [3], etc., the attacker usually exhibits an extraordinary large spread by contacting a large number of other hosts. Despite many efforts on detecting spreaders in network data streams [4], [5], [6], [7], production networks are still vulnerable under these attacks. One reason is that in many malicious attacks such as botnets [8] and stealthy DDoS attacks [9], the attacker prefers to reduce its fan-out by spreading the malicious communication over a long time span, so as to bypass the spreader detection. Although a number of methods for detecting persistent patterns in network data streams are proposed in recent years [10], [11], [12], [13], [14], however, they focus on detecting individual connections or sessions rather than attackers.

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61672486 and Grant 62072425. *(Corresponding author: Ye Tian.)*

The authors are with Anhui Key Laboratory on High Performance Computing, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anui, 230026, China (e-mail: wang_zh@mail.ustc.edu.cn; yetian@ustc.edu.cn; wyw0530@mail.ustc.edu.cn; szcw33@mail.ustc.edu.cn; fengzhi27@mail.ustc.edu.cn; xinming@ustc.edu.cn).

To address this problem, people consider to detect network flows that have abnormally large numbers of persistent elements, and refer to these flows as persistent-spreaders [9], [15], [16]. In these works, time is divided into *epochs*, a source or destination address is called a *flow*, and a distinct source-destination address pair is called an *element* in the flow. Given an element that includes all the packets sent from address $x$ to address $y$, it is considered as a persistent element if it has at least one packet that appears in no less than $k$ out of the $t$ recent epochs. For a flow composed of many elements, if the number of its persistent elements exceeds a threshold, it is classified as a persistent-spreader.

The above definition on a network flow's persistent-spread and existing solutions [9], [15], [16] for detecting persistent-spreaders following this definition have two limitations. First, when estimating an element's persistence, only the occurrences in the $t$ recent epochs are considered. However, as we will discuss in this paper, when an attacker is aware of the parameter values of $t$ and $k$, it can manipulate the contact pattern of each element to bypass the detection. Moreover, in these works, a persistent-spreader is defined as a flow that has enough number of persistent elements. However, an attacker can make a tradeoff between persistence and spread to bypass the detection. For example, a stealth DDoS attacker can deliberately reduce the number of its attacking machines in each epoch below the threshold, but instruct each machine to persistently consume resource of the target over a long time span. Similarly, a super-spreader can recruit a large number of attacking machines to quickly bring down the target within a short time. Obviously in both cases, the attacker can avoid being detected as a persistent-spreader.

In this paper, we present new definitions on an element's persistence and a network flow's persistent-spread, and following our definitions, we develop *PS-Sketch*, a system for estimating persistent-spreads of network flows and detecting persistent-spreaders in high-speed networks. Unlike the conventional definition that only considers the $t$ recent epochs, we define an element's *persistence* as the time-decaying sum of all its occurrences in its entire history. Based on the definition of element persistence, we define a flow's *persistent-spread* as the sum of the persistence of all the elements that belong to the flow, and consider a flow as a *persistent-spreader* if its persistent-spread exceeds a threshold. Comparing with the conventional definition, our definition of persistent-spreader covers a wide range of network flows from super-spreaders that have large numbers of non-persistent elements, to stealthy DDoS attackers that have moderate numbers of highly persis-

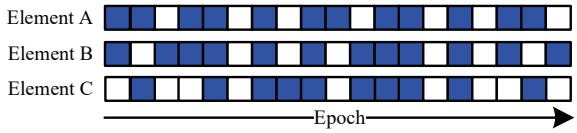



Fig. 1. Occurrences of three elements in a period of 19 epochs.

tent elements, as well as any flows between the two extremes, thus is difficult to bypass.

PS-Sketch adopts a sketch-based approach by processing packets with two adjacent sketch data structures, namely the persistent-sketch (P-sketch) and the spread-sketch (S-sketch), within a switch's pipeline. In the P-sketch, we exploit the low-pass filter (LPF) of programmable switch [17] to compute the time-decaying element persistence. In addition, for each element, we embed its estimated persistence in its first packet of each epoch to update the S-sketch, in which we employ the HyperLogLog (HLL) algorithm [18] as the distinct counter. To estimate a flow's persistent-spread, we integrate an element's persistence to the spread estimation of the flow that the element belongs to. In particular, we extend the HLL algorithm to encode the estimated persistence to the hash string of the element, and use the string to update the HLL registers.

From theoretical analysis, trace-driven evaluation, and hardware prototyping, we find that PS-Sketch has the following merits. First, PS-Sketch achieves high accuracy in estimating network flows' persistent-spreads. Second, PS-Sketch is capable to detect persistent-spreader flows accurately and outperforms the existing solutions. Third, by reporting the S-sketch at the end of each epoch, PS-Sketch is able to detect persistent-spreaders in a timely manner. Fourth, PS-Sketch is compatible with the commodity hardware switch and is able to process packets at line-rate. In the design, analysis, and evaluation of PS-Sketch, we make the following contributions.

- We present formal definitions of element persistence and network flow's persistent-spread, and show that our definitions better capture a persistent-spreader flow's behavior, and are difficult for attackers to bypass.
- We design PS-Sketch, a system for estimating persistent-spreads of network flows and detecting persistent-spreaders in network data streams.
- We present theoretical analysis on PS-Sketch, and obtain error bound on the estimating accuracy in respect to the memory usages of different components in the system.
- We show with trace-driven simulation that PS-Sketch achieves high accuracy in estimating persistent-spreads of network flows, detects persistent-spreaders in a timely manner, and outperforms the existing solutions.
- We prototype PS-Sketch in P4 on the Intel Tofino chip [19], and show that it is feasible to deploy PS-Sketch on commodity hardware switches. We make the prototype source code publicly available[1].

The remainder part of this paper is organized as follows. We discuss background and motivation in Sec. II; Sec. III presents the design and analysis of PS-Sketch; We evaluate PS-Sketch in Sec. IV, and discuss alternative design choices in Sec. V. Sec. VI discusses related works and we conclude in Sec. VII.

[1]https://github.com/wyw0530/PS-Sketch

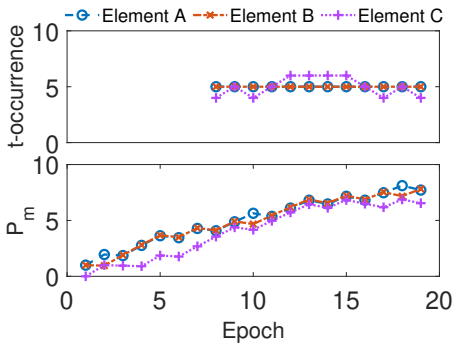



Fig. 2. Top: $t$-occurrences of three elements in Fig. 1 with $t = 8$; Bottom: $P_m$ of three elements with $\gamma = 0.05$.

## II. Background and Motivation

### A. Definitions of Persistence and Persistent-Spread

Consider a data stream of network packets, where we define a unique pair of source and destination addresses $m = (x, y)$ as an *element*, which contains all the packets sent from the source host $x$ to the destination $y$. We further group the elements with $x$ as source address as a *per-source flow*, which contains all the packets sent from $x$. Similarly, elements with $y$ as destination address form a *per-destination flow* that contains all the packets sent to $y$.

Time is divided into non-overlapping and contiguous time windows called *epochs*. Given an element $m$ from $x$ to $y$, people used to define its persistence as how many epochs the element occurs in the past $t$ epochs [10], [9], [11], [15], [12], [13], [16], [14], and we refer to such a definition as *$t$-occurrence persistence* in this paper.

Given a network flow $x$ composed of many elements, its persistent-spread is defined as number of the distinct persistent elements in the flow. In particular, [9], [15] estimates a flow's persistent-spread as number of the elements in the flow that occur in all the $t$ recent epochs; and [16] generalizes the problem by estimating a flow's *$k$-persistent spread*, which is the number of the distinct elements that occur in at least $k$ ($k \leq t$) out of the $t$ recent epochs.

However, since only the $t$ recent epochs are considered, if an attacker is aware of the pre-specified $t$ and $k$ parameter values, it can manipulate its contact pattern to bypass the detection. In particular, the attacker can contact a destination in exactly $k-1$ epochs per each $t$ consecutive epochs, but still contacts the destination frequently. For example, suppose $t = 8$, then the elements $A$ and $B$ in Fig. 1 each has a $t$-occurrence of 5, despite that both elements occur 12 times in the 19 epochs, as we can see in the top-figure of Fig. 2. On the other hand, element $C$ has a $t$-occurrence of 6 in as many as 4 epochs, although it occurs only 10 times in the 19 epochs. If we set $k = 6$, then $C$ will be classified as a persistent element, but $A$ and $B$ will not. If an attacker follows the pattern of element $A$ or $B$ in contacting a target, then no matter how many elements the attacker flow contains, none of them will be classified as a

persistent element, and the flow will bypass the detection by having a zero $k$-persistent spread by definition.

### B. Detecting Persistent-Spreader with Programmable Switch

Existing works employ a hybrid method for detecting spreaders or persistent-spreaders [4], [9], [15], [16], [5], [6], where hardware maintains simple data structures such as bit arrays, and the complicated operations for constructing the estimators are conducted by software. However, as commodity switches nowadays provide a line-rate up to dozens even hundreds of Gbps, it is challenging for the software-based solutions to keep up with the speed of hardware switches.

In recent years, a wide range of network algorithms, which used to be implemented with software or dedicated hardware, have opportunities to be realized on hardware programmable switches (e.g., switches based on the Intel Tofino chip [19]) and run at line-rate. Obviously, in high-speed networks, it is ideal to estimate network flows' persistent-spreads with programmable switches. However, to fulfill such an objective, we need to deal with the following challenges.

- The first challenge is how to define persistence and persistent-spread in a way to accurately capture a persistent-spreader flow's behavior, and is difficult for attackers to bypass.
- The second challenge is how to design data structures and algorithms that are fast, accurate, and memory-efficient to estimate network flows' persistent-spreads in network data streams.
- Finally, unlike software, programmable switches have limited resources and many restrictions, such as the lack of support for float-point values, no support for multiplication, division, exponential and logarithmic operations, etc., it is challenging to overcome these limitations.

## III. Estimating Persistent-Spread with PS-Sketch

Motivated by the observation in Sec. II, in this work, we present *PS-Sketch*, a system for accurately estimating network flows' persistent-spreads with a sketch-based approach. Table I lists the notations frequently used in this paper.

### A. Problem Statement and System Overview

*1) Problem statement:* Given a network data stream, we consider a pair of unique source-destination address pair $m = (x, y)$ as an *element*, and group all the elements with $x$ as source address as a *per-source flow*, or a *flow* for short. Note that although we focus on per-source flows, PS-Sketch is also applicable to per-destination flows as well.

Time is divided into *epochs*. We denote the current epoch as $t_0$, and denote the previous epochs as $t_{-1}$, $t_{-2}$, $\cdots$. For an element $m = (x, y)$, we define its *persistence* $P_m$ as

$$P_m = I_{m,0} + I_{m,-1} \cdot e^{-\gamma} + I_{m,-2} \cdot e^{-2\gamma} + \cdots \quad (1)$$

where $e$ is Euler's number, $I_{m,k} = 1$ ($k = \cdots, -1, 0$) if at least one packet of element $m$ appears in epoch $t_k$, and $I_{m,k} = 0$ otherwise. Note that unlike the works that only count an element's occurrences in the $t$ recent epochs as its persistence (i.e., the $t$-occurrence persistence), (1) incorporates an element's occurrences during its entire history, and decays over time with a factor of $e^{-\gamma}$ per epoch, thus better captures the element's behavior. To show this, we present $P_m$ of the elements $A$, $B$, and $C$ in Fig. 1 in the bottom figure of Fig. 2. We can see that the elements $A$ and $B$, which occur more often than $C$, have their $P_m$ higher than $C$ all the time.

Note that although $P_m$ summarizes an element's occurrences in its entire history, however, its value is bounded by $\frac{e^\gamma}{e^\gamma - 1}$. In other words, a counter for $P_m$ will never overflow.

For a network flow $x$ containing many elements, at the end of each epoch, we define its *persistent-spread* $S_x$ as

$$S_x = \sum_{m \in x} P_m \cdot I_{m,0} \quad (2)$$

In addition, if a flow $x$ has its persistent-spread $S_x$ greater than a pre-specified threshold, it is considered as a *persistent-spreader*.

*Note that with our definitions in (1) and (2), the problem of detecting persistent-spreaders that we study in this paper is very different from the one studied in the previous works [9], [15], [16].* In particular, our definition of persistent-spread in (2) does not impose a threshold on individual element's persistence, but aggregates $P_m$ of all the elements in a flow to compute $S_x$. Since we only apply a threshold on $S_x$ to detect persistent-spreaders, either a flow with a large number of none-persistent elements (i.e., a super-spreader), or a flow with moderate numbers of highly persistent elements (i.e., a stealthy DDoS attacker), or any flow between the two extremes, may have its $S_x$ large enough to be detected as a persistent-spreader. In other words, it is difficult for an attacker to bypass the detection by making a tradeoff between element persistence and flow spread.

TABLE I
Frequently used notations

| Notation | Meaning |
|---|---|
| Defined in Sec. III-A | |
| $m = (x, y)$ | Element composed of packets from $x$ to $y$. |
| $x$ | Flow composed of packets sent from $x$. |
| $P_m$ | Real persistence of element $m$ as defined in (1). |
| $S_x$ | Real persistent-spread of flow $x$ as defined in (2). |
| Defined in Sec. III-B | |
| $\mathbf{B}$ | P-sketch composed of $d_1 \times w_1$ buckets. |
| $\mathbf{B}_i[j].C$, $\mathbf{B}_i[j].t$ | LPF and timestamp of the P-sketch bucket $\mathbf{B}_i[j]$. |
| $\hat{P}_m$ | Persistence of element $m$ estimated by P-sketch. |
| $p.C$, $p.F$ | Counter and flag associated with packet $p$. |
| $\varepsilon_1$ | Approximation parameter of the P-sketch. |
| $\delta_1$ | Error probability of the P-sketch. |
| Defined in Sec. III-C | |
| $H(m)$ | $L$-bit hash string of element $m$. |
| $H_1(m)$, $H_2(m)$ | $b$-bit and $(L-b)$-bit sub-strings of $H(m)$. |
| $\tilde{S}_x$ | Persistent-spread of flow $x$ approximated by HLL. |
| $\sigma$ | Error bound of HLL approximation. |
| Defined in Sec. III-D | |
| $\mathbf{C}$ | S-sketch composed of $d_2 \times w_2$ buckets. |
| $\mathbf{C}_i[j].H$ | HLL counter of the S-sketch bucket $\mathbf{C}_i[j]$. |
| $\mathbf{C}_i[j].X$, $\mathbf{C}_i[j].L$ | Flow ID and level of the S-sketch bucket $\mathbf{C}_i[j]$. |
| $\hat{S}_x$ | Persistent-spread of flow $x$ estimated by S-sketch. |
| $\varepsilon_2$ | Approximation parameter of the S-sketch. |
| $\delta_2$ | Error probability of the S-sketch. |

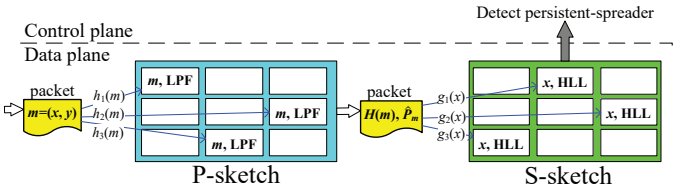



Fig. 3. Overview of PS-Sketch.

Given our definitions on persistence and persistent-spread, we pursue two objectives in the design and development of PS-Sketch: 1) the system should be able to estimate a network flow's persistent-spread as defined in (2) accurately; 2) the solution should be applicable in high-speed networks, and in particular, be deployable on hardware switches.

*2) System overview:* Fig. 3 presents an overview of PS-Sketch, which is composed of two sketch data structures, namely the *persistence sketch* (referred to as the *P-sketch*) and the *spread sketch* (referred to as the *S-sketch*).

As shown in Fig. 3, both the P-sketch and the S-sketch are updated by packets traversing the switch pipeline. When a packet of element $m$ and flow $x$ arrives to the switch, it first traverses the P-sketch and has its element persistence estimated as $\hat{P}_m$. If the packet is the first packet of the element in the current epoch, it encodes $\hat{P}_m$ to the hash string of its element ID, and uses the hash string to update the S-sketch for estimating the flow's persistent-spread $\hat{S}_x$. At the end of each epoch, the control plane detects persistent-spreader flows from the S-sketch reported by the switch data plane.

### B. Estimating Element Persistence with P-sketch

*1) Data structure and operation:* We extend the Count-Min (CM) sketch [20] to design the P-sketch $\mathbf{B}$, which has $d_1$ bucket arrays, and each array contains $w_1$ buckets. The sketch is associated with $d_1$ mutually independent hash functions $h_i(\cdot) : \mathbb{E} \to \{1, \cdots, w_1\}$, where $\mathbb{E}$ is the element ID universe. Each sketch bucket has two fields: a *low-pass filter* (*LPF*) and a timestamp. LPF is a self-decaying counter. More specifically, if we add value $a$ at the current time $t$ to an LPF with previous value $v'$ and last update time $t'$ ($t' \leq t$), then its new value will become

$$v = a + v' \cdot e^{-(t-t')/\tau} \tag{3}$$

where $\tau$ is the decay time parameter. By setting $\tau = 1/\gamma$, it is easy to see that LPF can be used for tracking an element's time-decaying persistence $P_m$ as defined in (1). Note that LPF is natively supported as an advanced feature of the Intel Tofino chip [17], and it can also be approximated using basic P4 on hardware programmable switches without dedicated support [21], [22]. We use $\mathbf{B}_i[j].C$ to denote the LPF of the P-sketch bucket $\mathbf{B}_i[j]$, and use $\mathbf{B}_i[j].|C|$ to denote its counter value.

On receiving a packet $p$ that belongs to an element $m$, the switch allocates a counter in the metadata associated with $p$, denoted as $p.C$, and uses it to estimate $m$'s persistence. Initially, $p.C$ is assigned with a value slightly larger than $\frac{e^\gamma}{e^\gamma - 1}$. The packet $p$ also has a flag $p.F$ initialized as $False$ in its metadata, which indicates whether $p$ is the first packet of the element $m$ during the current epoch. The switch applies the hash functions $h_i(\cdot)$ to map packet $p$ to the buckets $\mathbf{B}_i[h_i(m)]$, $i = 1, \cdots, d_1$, and processes the packet as the following:

- If $\mathbf{B}_i[h_i(m)].t$, which is the timestamp of the bucket, is not in the current epoch $t_0$, set $\mathbf{B}_i[h_i(m)].t$ as the current time, set $p.F$ as $True$, and add 1 to $\mathbf{B}_i[h_i(m)].C$.
- Compare $p.C$ with $\mathbf{B}_i[h_i(m)].|C|$, and if $p.C > \mathbf{B}_i[h_i(m)].|C|$, assign $\mathbf{B}_i[h_i(m)].|C|$ to $p.C$.

**Algorithm 1:** Packet processing with P-sketch

```
Algorithm
    Input : A packet p belonging to element m.
    p.C ← a value greater than e^γ/(e^γ−1);
    p.F ← False;
    for i = 1, ··· , d1 do
        if B_i[h_i(m)].t ∉ t0 then
            Add(B_i[h_i(m)].C,1);
            B_i[h_i(m)].t ← curr_time; p.F ← True;
        if p.C > B_i[h_i(m)].|C| then
            p.C ← B_i[h_i(m)].|C|;
```

Algorithm 1 presents the procedure of packet processing with the P-sketch. Note that after packet $p$ has accessed all the $d_1$ mapped buckets, $p.C$ is indeed the element $m$'s estimated persistence $\hat{P}_m$, i.e.,

$$p.C = \hat{P}_m = \min(\mathbf{B}_i[h_i(m)].|C|; i = 1, \cdots, d_1) \tag{4}$$

Fig. 4 presents some examples. In the cases of Fig. 4(a) and (b), each packet is the first packet of its element in the current epoch, and after updating the P-sketch, it carries the minimal of the LPF counter values in $p.C$, and has the flag $p.F$ set as $True$, which means that the packet will be subsequently used to update the S-sketch. In Fig. 4(c), the packet is not the first packet of its element in the current epoch, or there exist hash collisions at all its mapped buckets, and in both cases, the packet's flag $p.F$ remains as $False$.

*2) Analysis:* For $\hat{P}_m$ estimated by the P-sketch as in (4), we have the following result regarding its estimating accuracy.

**Theorem 1.** *For $w_1 = \lceil e/\varepsilon_1 \rceil$ and $d_1 = \lceil \ln(1/\delta_1) \rceil$, with a probability at least $1 - \delta_1$, $\hat{P}_m$ is no larger than $P_m + \varepsilon_1 P$, where $P = \sum_m P_m$. In other words, we have*

$$\Pr[\hat{P}_m \leq P_m + \varepsilon_1 P] \geq 1 - \delta_1 \tag{5}$$

The proof of Theorem 1 is given in Appendix A-A.

Fig. 4(c) shows that for an element $m$ that occurs in an epoch, when its first packet encounters hash collisions at all the mapped buckets, the packet will have $p.F = False$, and in this case, the element $m$ fails to update the S-sketch.

To analyze the probability of such an incident, let $\mathbb{M}_k$ be the set of the distinct elements in epoch $t_k$ ($k = \cdots, -1, 0$), then the probability that an element encounters a hash collision at a mapped bucket is $1 - (1 - \frac{1}{w_1})^{|\mathbb{M}_k|} \approx \frac{|\mathbb{M}_k|}{w_1}$, and the probability that hash collisions happen at all its mapped buckets is $(\frac{|\mathbb{M}_k|}{w_1})^{d_1}$, which is close to 0 when $w_1 >> |\mathbb{M}_k|$. In other words, for each element in the data stream, after being

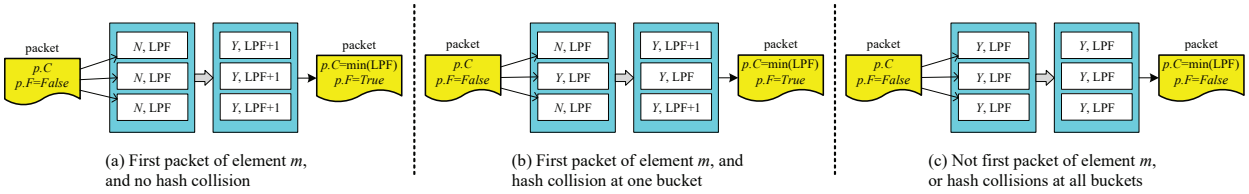



Fig. 4. Examples of packet processing with P-sketch, where 'Y' or 'N' indicates whether timestamp of the bucket is in the current epoch or not.

processed by the P-sketch, its first packet has $p.F = True$ and updates the S-sketch with a probability close to 1.

### C. Integrating Persistence to HyperLogLog

*1) HyperLogLog preliminary:* Distinct counter is a powerful tool for estimating cardinality (i.e., number of distinct items) of a data stream, and representative distinct counters include PCSA [23], LogLog [24], HyperLogLog (HLL) [18], multi-resolution bitmap [25], etc. In this work, we choose HLL in the S-sketch because of its memory efficiency.

We briefly introduce HLL as the following. An HLL counter is composed of $s = 2^b$ registers, denoted as $M[i]$, $i = 1, \cdots, s$, which are initialized as 0. Let $H(\cdot) : \mathbb{E} \to \{0,1\}^L$ be a hash function that hashes an element ID to an $L$-bit hash string $H(m)$, where $L > b$. To update the registers, $H(m)$ is divided into two sub-strings: the first $b$ bits is denoted as $H_1(m)$ and the remaining $L - b$ bits is denoted as $H_2(m)$. Let $\rho(H_2(m))$ be the position of the left-most '1' in $H_2(m)$, then the register $M[H_1(m)]$ is updated as

$$M[H_1(m)] = \max(M[H_1(m)], \rho(H_2(m))) \tag{6}$$

For example, suppose $H(m)$ ='11000110', $L = 8$, $b = 2$, and by dividing $H(m)$, we have $H_1(m)$ ='11'= 3 and $\rho(H_2(m)) = 4$. If the original value of the register $M[3]$ is smaller than 4, then the element $m$ updates $M[3]$ to 4.

After being updated by all the elements in a data stream, the stream's cardinality is estimated based on the harmonic mean of $\{2^{M[i]}\}_{i=1,\cdots,s}$ as

$$\alpha_s \cdot s^2 \cdot (\sum_{i=1}^{s} 2^{-M[i]})^{-1} \tag{7}$$

where $\alpha_s$ is the bias correction constant. In particular, $\alpha_{16} = 0.673$, $\alpha_{32} = 0.697$, $\alpha_{64} = 0.709$, etc. Finally, the cardinality estimation $\hat{D}$ is computed based on the raw value from (7) and number of the registers equaling to 0. In addition, given the real cardinality $D$, $D(1-\sigma) \le \hat{D} \le D(1+\sigma)$, where $\sigma$ is multiples of $\frac{1.04}{\sqrt{s}}$ [18].

*2) Integrating $\hat{P}_m$ to HLL:* The rationale behind HLL is to use randomization to approximate cardinality. Randomization is achieved with the hash function $H(\cdot)$, where intuitively a hash string with more leading '0's is less likely and indicates a larger cardinality. If a bit pattern $0^{(\rho-1)}1*$ is observed in $H_2(m)$, then a good cardinality approximation should be $2^\rho$.

Our basic idea is to encode an element's persistence into its hash string. Suppose that the HLL encounters an element $m$ with its persistence $\hat{P}_m$ estimated by the P-sketch, after applying the hash function $H(\cdot)$ and obtaining the two sub-strings of $H_1(m)$ and $H_2(m)$, we right-shift the sub-string $H_2(m)$ $\lceil \log_2 \hat{P}_m \rceil$ bits, and denote the resulting sub-string as $H_2'(m)$. Intuitively, if the original bit pattern $0^{(\rho-1)}1*$ of $H_2(m)$ suggests a spread of $2^\rho$, then by right-shifting $\lceil \log_2 \hat{P}_m \rceil$ bits, $H_2'(m)$ indicates a spread of $2^{(\rho+\lceil \log_2 \hat{P}_m \rceil)}$, whose value is between $\hat{P}_m \cdot 2^\rho$ and $\hat{P}_m \cdot 2^{\rho+1}$. In other words, we multiply $\hat{P}_m$ with the spread approximation. After the bit-shifting, the sub-strings $(H_1(m), H_2'(m))$ update the HLL registers as

$$M[H_1(m)] = \max(M[H_1(m)], \rho(H_2'(m))) \tag{8}$$

For example, when $H(m)$ ='11000110', $L = 8$, $b = 2$, and $\hat{P}_m = 3$, we have $\lceil \log_2 \hat{P}_m \rceil = 2$, and after the bit-shifting, $H_2'(m)$ ='000001' and $\rho(H_2'(m)) = 6$.

*3) Analysis:* We analyze the errors introduced by bit-shifting and HLL approximation in this section. Suppose that for a flow $x$, one dedicated HLL counter is used to approximate its persistent-spread as above described, and we denote the result as $\tilde{S}_x$. Note that $\tilde{S}_x$ contains the errors introduced by P-sketch, bit-shifting, and HLL approximation, and for the accuracy of $\tilde{S}_x$, we have the following result.

**Theorem 2.** *For a network flow $x$, $\tilde{S}_x \ge (1-\sigma)S_x$, and with a probability at least $1-\delta_1$, we have*

$$\tilde{S}_x \le 2(1+\sigma)(S_x + \varepsilon_1 P D_x) \tag{9}$$

*where $D_x = \sum_{m \in x} I_{m,0}$ is flow $x$'s spread in epoch $t_0$.*

The proof of Theorem 2 is given in Appendix A-B.

### D. Estimating Flow Persistent-Spread with S-sketch

*1) Data structure and operation:* During each epoch, after being processed by the P-sketch, an element's first packet is further processed by the S-sketch (by matching $p.F$ with $True$). The S-sketch $\mathbf{C}$ is composed of $d_2$ bucket arrays with each array containing $w_2$ buckets. Each bucket, say $\mathbf{C}_i[j]$, has three fields: the field $\mathbf{C}_i[j].H$ is an HLL distinct counter, whose value is denoted as $\mathbf{C}_i[j].|H|$; the field $\mathbf{C}_i[j].L$ is the level of this bucket, which records the maximum number of the leading '0's in $H_2'(m)$ that the bucket has ever encountered; and the field $\mathbf{C}_i[j].X$ keeps the ID of the flow that most recently updates $\mathbf{C}_i[j].L$. The S-sketch is associated with $d_2$ mutually independent hash functions $g_i(\cdot) : \mathbb{F} \to \{1, \cdots, w_2\}$, $i = 1, \cdots, d_2$, where $\mathbb{F}$ is the flow ID universe.

Algorithm 2 presents the procedure for updating the S-sketch. After receiving a packet with its element ID as $m$ and flow ID as $x$, which has $p.F = True$ and carries $m$'s persistence estimation $\hat{P}_m$ in $p.C$, the switch computes the hash string $H(m)$, right-shifts $\lceil \log \hat{P}_m \rceil$ bits to obtain the two sub-strings of $H_1(m)$ and $H_2'(m)$, and uses them to update the

**Algorithm 2:** Packet processing with S-sketch

```
Algorithm
    Input : Packet p of element m and flow x with
            p.F = True and carrying P̂_m in p.C.
    Compute (H_1(m), H'_2(m)) with m and P̂_m;
    for i = 1, ··· , d_2 do
        Update C_i[g_i(x)].H with (H_1(m), H'_2(m));
        if C_i[g_i(x)].L < ρ(H'_2(m)) − 1 then
            C_i[g_i(x)].L ← ρ(H'_2(m)) − 1;
            C_i[g_i(x)].X ← x;
```

S-sketch. In particular, for each bucket array, the switch applies the corresponding hash function $g_i(\cdot)$ to map the packet to the bucket $\mathbf{C}_i[g_i(x)]$, $i = 1, \cdots, d_2$, and updates $\mathbf{C}_i[g_i(x)].H$, the HLL counter of the bucket, with $(H_1(m), H_2'(m))$ as in (8). The switch further compares the number of the leading '0's in $H_2'(m)$ (i.e., $\rho(H_2'(m)) - 1$) with $C_i[g_i(x)].L$: if $C_i[g_i(x)].L$ is smaller, assign $\rho(H_2'(m)) - 1$ to $\mathbf{C}_i[g_i(x)].L$, and assign $x$ to $\mathbf{C}_i[g_i(x)].X$.

At end of each epoch, the switch reports the S-sketch $\mathbf{C}$ to the control plane, which checks all the buckets in $\mathbf{C}$. If a bucket has an HLL counter value $\mathbf{C}_i[j].|H|$ exceeding a pre-specified threshold, we obtain the corresponding flow ID as $x = \mathbf{C}_i[j].X$, and estimate the persistent-spread of flow $x$ as

$$\hat{S}_x = \min(\mathbf{C}_i[g_i(x)].|H|; i = 1, \cdots, d_2) \tag{10}$$

If $\hat{S}_x$ still exceeds the threshold, then flow $x$ is classified as a persistent-spreader.

*2) Analysis:* Recall that if we use one dedicated HLL counter to estimate each flow, then for a flow $x$, its persistent-spread estimated by the counter is $\tilde{S}_x$. However, in the S-sketch, an HLL counter is shared among multiple flows, so comparing with $\tilde{S}_x$, the S-sketch introduces additional errors. For a flow $x$'s persistent-spread $\hat{S}_x$ estimated by the S-sketch as in (10), we have the following result regarding its accuracy.

**Theorem 3.** *For $w_2 = \lceil e/\varepsilon_2 \rceil$ and $d_2 = \lceil \ln(1/\delta_2) \rceil$, with a probability at least $1 - \delta_2$, $\hat{S}_x$ is no larger than $\tilde{S}_x + \varepsilon_2 \tilde{S}$, where $\tilde{S} = \sum_x \tilde{S}_x$. In other words, we have*

$$\Pr[\hat{S}_x \leq \tilde{S}_x + \varepsilon_2 \tilde{S}] \geq 1 - \delta_2 \tag{11}$$

The proof of Theorem 3 is given in Appendix A-C.

Finally, combining Theorems 2 and 3, we have the following result regarding the upper error bound of flow $x$'s estimated persistent-spread $\hat{S}_x$ comparing to its real persistent-spread $S_x$.

**Theorem 4.** *Given $\varepsilon_1$, $\varepsilon_2$, $\delta_1$, $\delta_2$, and $\sigma$, with a probability at least $(1 - \delta_1)(1 - \delta_2)$, we have*

$$\begin{aligned} \hat{S}_x \quad \leq \quad & 2(1+\sigma)S_x + 2\varepsilon_1(1+\sigma)PD_x \\ & + 2\varepsilon_2(1+\sigma)S + 2\varepsilon_1\varepsilon_2(1+\sigma)PD \end{aligned} \tag{12}$$

*where $P = \sum_m P_m$, $S = \sum_x S_x$, $D = \sum_x D_x$, and $D_x = \sum_{m \in x} I_{m,0}$ is flow $x$'s spread in epoch $t_0$.*

The proof of Theorem 4 is given in Appendix A-D.

From Theorem 4, we can see that the inaccuracy of a network flow's persistent-spread estimation come from four sources: 1) hash collisions in the P-sketch, 2) bit-shifting when integrating the estimated persistence $\hat{P}_m$ to $H(m)$, 3) HLL approximation, and 4) hash collisions in the S-sketch. In the subsequent section, we evaluate the impacts of different components on PS-Sketch's estimating accuracy.

## IV. Evaluation

We implement PS-Sketch with Python on a machine with an 8-core Intel i7-8700 CPU, and evaluate it using traffic traces captured from real-world networks. With the trace-driven simulation, we find that 1) PS-Sketch is capable to estimate a network flow's persistent-spread accurately; 2) PS-Sketch is efficient in memory utilization, enlarging the P-sketch and the S-sketch is favorable to improving the estimating accuracy, but enlarging the HLL counters by employing more registers is not necessarily helpful; 3) PS-Sketch can detect persistent-spreader flows with high accuracy in a timely manner, and outperforms the existing solutions [16], [7].

We also implement a PS-Sketch prototype in P4 on an Edgecore Wedge 100BF-32X programmable switch based on the Intel Barefoot Tofino chip [19]. We show that PS-Sketch is inexpensive in resource consumption, and is deployable in high-speed networks.

### A. Experiment Setup

We use the following real-world network traffic traces in our evaluation.

- *MAWI trace*: The MAWI Working Group [26] captures IP traffic from ISP backbone links. Each trace lasts 15 minutes, and we select one trace, which contains $145,090$ distinct per-source flows and $8,892,143$ distinct elements, for our evaluation. We divide the trace into 15 epochs, with each epoch lasting one minute.
- *Facebook (FB) trace*: This is a 24-hour trace captured from machines in a cluster of Facebook [27], [28]. Each epoch lasts one minute, and there are $1,449$ epochs in the trace. The trace contains $6,130$ distinct flows and $1,125,159$ distinct elements.
- *Witty trace*: The UCSD Network Telescope [29] collects packets sent from hosts infected by the Witty worm in 2004 [30]. We select a trace lasting 60 minutes, and divide the trace into 60 epochs, with each epoch lasting one minute. There are $2,741$ distinct flows and $9,253,979$ distinct elements in the trace.

Different traces have different characteristics. The MAWI trace contains many super-spreader flows that have large spreads, and it also contains a few flows that are composed of moderate numbers of very persistent elements (e.g., elements occurring in all the 15 epochs). Flows in the FB trace have moderate spreads due to limited destinations within the cluster, but as the trace lasts $1,449$ epochs, many elements are highly persistent by occurring in hundreds of epochs. For the Witty trace, since it contains only the worm traffic sent from infected hosts, the trace is dominated with supper-spreader flows.

We configure PS-Sketch as the following. The P-sketch contains $d_1 = 3$ arrays of buckets, and each bucket has a 32-bit LPF counter and a 32-bit timestamp; the S-sketch has $d_2 = 4$ bucket arrays, and each bucket is composed of an HLL counter, a 32-bit flow ID, and a 5-bit level. For configuring the HLL counters, we fix $L = 32$, and set the size of each register as 5 bits. We vary $b$, $w_1$, and $w_2$ to change the sizes of the HLL counters, the P-sketch, and the S-sketch respectively in our evaluation. More specifically, in the experiments with the MAWI and the Witty traces, we apply a default configuration containing a P-sketch of 5 MB and an S-sketch of 1 MB, and set $b = 4$ for the HLL counters. We denote the configuration with a three-tuple as (5 MB−4−1 MB). In the experiments with the FB trace, the default PS-Sketch configuration, denoted as (1 MB−4−256 kB), contains a 1 MB-sized P-sketch and a 256 kB-sized S-sketch, and we set $b = 4$ for the HLL counters. If not otherwise specified, we set $\gamma = 0.05$ for the MAWI and the Witty traces and set $\gamma = 0.1$ for the FB trace.

We focus on the following metrics in our evaluation.

- *Averaged relative error (ARE)*: ARE is computed as

$$ARE = \frac{1}{n} \sum_{x} \frac{|\hat{S}_x - S_x|}{S_x} \tag{13}$$

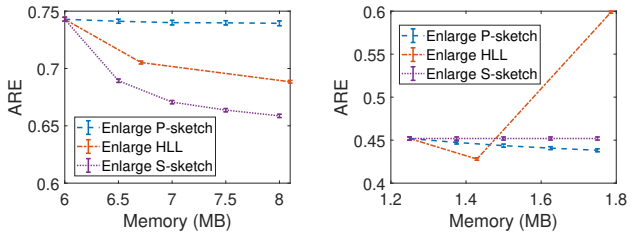
  where $n$ is number of the estimations, $S_x$ and $\hat{S}_x$ are flow $x$'s real and estimated persistent-spreads.
- *Precision*: Precision is defined as the ratio between true positives and the sum of true positives and false positives. In detecting persistent-spreaders, it is the probability that a detected flow is a real persistent-spreader.
- *Recall*: Recall is defined as the ratio between true positives and the sum of true positives and false negatives. In detecting persistent-spreaders, it is the probability that a real persistent-spreader flow gets detected.
- *Detecting time and detecting time difference (DTD)*: We define the earliest epoch that a flow is detected as its detecting time. We also evaluate DTD, which is defined as the difference between the detecting time of a flow and the epoch that the flow's real persistent-spread first exceeds the threshold. Detecting time and DTD reflect how timely a persistent-spreader is detected.

We do not consider the F1 score, which is the geometric mean of precision and recall, in our evaluation. Note that in network security, recall should be more important than precision, as missing a real persistent-spreader would bring a larger potential damage than mis-classifying an innocent flow, thus we believe that the two metrics should not be averaged.

### B. Estimating Persistent-Spread

In this section, we employ the MAWI and the FB traces to evaluate PS-Sketch. We start with the default configurations, and enlarge the P-sketch, the HLL counters, and the S-sketch respectively to investigate the their impacts on the estimating accuracy in estimating network flows' persistent-spreads.

*1) Impact of HLL counter size:* We first increase the number of the registers in each HLL counter by increasing $b$ from 4 to 6. Note that as each S-sketch bucket contains an HLL counter, the size of the S-sketch also increases with $b$. For estimating persistent-spreads of the flows in the MAWI trace, we evaluate PS-Sketch with a series of configurations from (5 MB−4−1 MB) to (5 MB−6−3.14 MB). Similarly, to estimate persistent-spreads of the flows in the FB trace, we evaluate with the configurations from (1 MB−4−256 kB) to (1 MB−6−805 kB).




(a) MAWI (b) FB

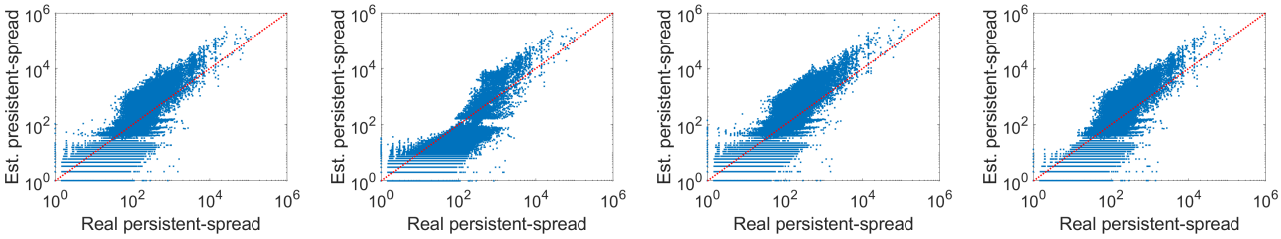
Fig. 5. AREs of persistent-spread estimations under various PS-Sketch configurations that have different memory usages with different-sized P-sketches, HLL counters, and S-sketches respectively.

Fig. 5 presents AREs of the persistent-spread estimations under the different configurations. From the figures, we can see that in the MAWI trace, employing more registers can moderately reduce the errors, but to our surprise, when $b$ is increased to 6, a larger ARE is observed with the FB trace.

We explain the paradoxical observation in Fig. 5 with the *small-range correction* of the HLL algorithm. In HLL, when the cardinality of a data stream is small, after being updated by all the elements, some registers may still have a value of 0 as initialized. If the harmonic mean computed by (7) is no larger than $\frac{5}{2}s$, where $s = 2^b$ is the number of the registers, the algorithm directly returns $s \log(s/V)$ as the estimation, in which $V$ is the number of the registers equaling to 0 [18].

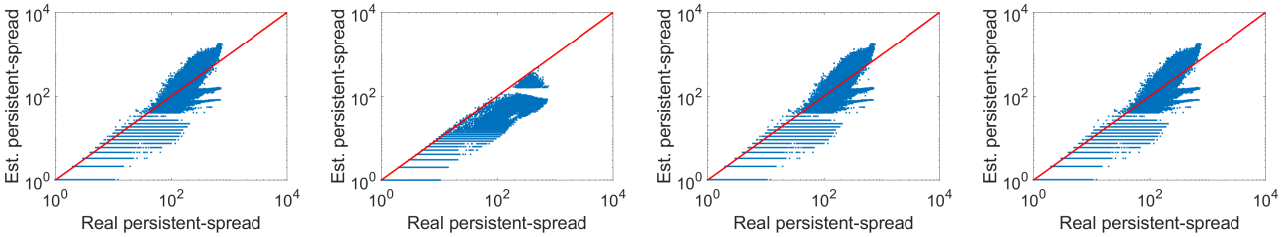
In PS-Sketch, since we encode an element's persistence before updating the HLL counters, if a flow contains a moderate number of elements, even though the elements are of high persistence, each element updates the HLL counters only once, and at the end of each epoch, some registers in a counter are likely to be 0. In this case, the small-range correction is triggered, and the HLL counter will under-estimate the flow's persistent-spread as $s \log(s/V)$, which indeed has nothing to do with the elements' persistence. Moreover, since $s = 2^b$, when $b$ increases, more flows will have their harmonic means computed by (7) smaller than $\frac{5}{2}s$, and as a consequence, their persistent-spreads will be under-estimated.

With the above observation, we analyze the estimating errors in the MAWI trace as follows. Since many persistent-spreaders in the trace are also super-spreaders in the trace, when estimating their persistent-spreads, the small-range correction of the HLL algorithm is unlikely to be triggered, and using more registers actually improves accuracy of the HLL approximation. To show this, in Fig. 6(a) and (b), we present the scatter plots of the real and estimated persistent-spreads of the flows in the MAWI trace under the PS-Sketch configurations of (5 MB−4−1 MB) and (5 MB−6−3.14 MB) respectively. We can see that by using 4 times more registers per HLL counter, the errors are reduced, especially for the




(a) (5 MB−4−1 MB) (b) (5 MB−6−3.14 MB) (c) (7 MB−4−1 MB) (d) (5 MB−4−3 MB)

Fig. 6. Scatter plots of real and estimated persistent-spreads of flows in MAWI trace under different configurations.




(a) (1 MB−4−256 kB) (b) (1 MB−6−805 kB) (c) (1.5 MB−4−256 kB) (d) (1 MB−4−768 kB)

Fig. 7. Scatter plots of real and estimated persistent-spreads of flows in FB trace under different configurations.

flows with large persistent-spreads.

On the other hand, as the FB trace lasts $1,449$ epochs, many flows have a moderate number of highly persistent elements, and after using more registers by increasing $b$, these flows are likely to be impacted by the small-range correction of the HLL algorithm and have their persistent-spreads under-estimated. To show this, we present the scatter plots of the real and estimated persistent-spreads of the flows under the configurations of (1 MB−4−256 kB) and (1 MB−6−805 kB) in Fig. 7(a) and (b) respectively. One can see that after using more registers in each HLL counter, nearly all the flows are under-estimated, which explains the increased ARE as observed in Fig. 5(b).

*2) Impact of P-sketch size:* We then enlarge the P-sketch to examine its impact on the estimating accuracy. More specifically, for estimating in the MAWI trace, we increase $w_1$ to enlarge the P-sketch from 5 MB to 7 MB, and evaluate with a series of PS-Sketch configurations from (5 MB−4−1 MB) to (7 MB−4−1 MB). For the FB trace, we enlarge the P-sketch from 1 MB to 1.5 MB, and experiment with the configurations from (1 MB−4−256 kB) to (1.5 MB−4−256 kB).

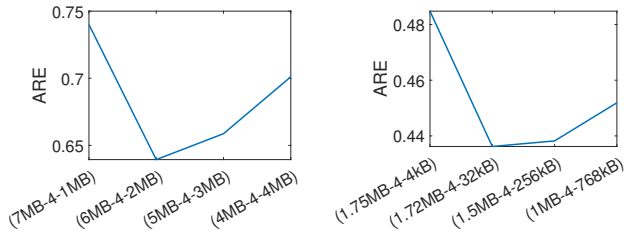
Fig. 5 presents AREs of the persistent-spread estimations achieved under the configurations containing different-sized P-sketches. We can see that for the MAWI trace, enlarging the P-sketch is the least effective way to reduce the errors, but it is the most effective approach for the FB trace. This is because in the MAWI trace which lasts only 15 epochs, many persistent-spreaders detected by PS-Sketch are super-spreaders with low element persistence, and enlarging the P-sketch for improving the persistence estimating accuracy is not very helpful. To show this, we present the scatter plots of the real and estimated persistent-spreads of the flows under the configurations of (5 MB−4−1 MB) and (7 MB−4−1 MB) in Fig. 6(a) and (c) respectively, and find that there is no obvious difference between them.

On the other hand, since the FB trace contains as many as $1,449$ epochs, with a higher persistence, an element has a greater impact on the persistent-spread of the flow that it belongs to, and enlarging the P-sketch is effective for reducing the estimating errors. To show this, we compare the scatter plots of the real and estimated persistent-spreads of the flows under the configurations of (1 MB−4−256 kB) and (1.5 MB−4−256 kB) in Fig. 7(a) and (c), and one can see that by enlarging the P-sketch, the errors in the persistent-spread estimations are reduced.

*3) Impact of S-sketch size:* We also enlarge the S-sketch by increasing $w_2$, and investigate the impact on the estimating accuracy. For estimating persistent-spreads of the flows in the MAWI trace, we evaluate with a series of PS-Sketch configurations from (5 MB−4−1 MB) to (5 MB−4−3 MB), and for the FB trace, we experiment with the configurations from (1 MB−4−256 kB) to (1 MB−4−768 kB). From Fig. 5, we can see that in the MAWI trace, enlarging the S-sketch is the most effective way to reduce the errors, but it has little effect on the FB trace.

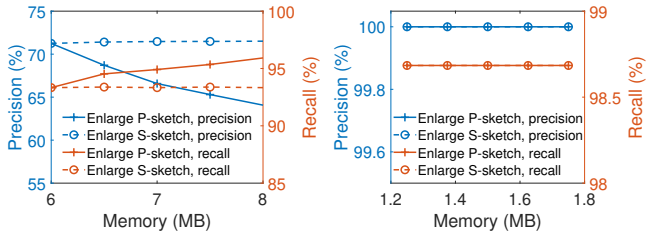
We explain the observations with the differences of the two traces: The MAWI trace contains over one hundred thousand flows, whose persistent-spreads are over-estimated when the size of the S-sketch is limited, and enlarging the S-sketch helps to reduce the errors. To show this, we compare the scatter plots of the real and estimated persistent-spreads of the flows under the configuration of (5 MB−4−1 MB) and (5 MB−4−3 MB) in Fig. 6(a) and (d), and one can see that by enlarging the S-sketch, the estimating errors are reduced.

On the contrary, since there are relatively fewer flows in the FB trace, a small-sized S-sketch can achieve a good accuracy, and further enlarging it does not bring significant benefit. To show this, we compare the scatter plots of the real and estimated persistent-spreads of the flows under the




(a) MAWI (b) FB

Fig. 8. AREs of persistent-spread estimations under various PS-Sketch configurations given fixed total memory usage.




(a) MAWI (b) FB

Fig. 9. Precisions and recalls for detecting persistent-spreaders in MAWI and FB traces under various PS-Sketch configurations.

configurations of (1 MB−4−256 kB) and (1 MB−4−768 kB) in Fig. 7(a) and (d), and find that there are no obvious reductions on the errors.

*4) Fixed total memory usage:* Finally, we consider the case that the P-sketch and the S-sketch consume a fixed total amount of memory, and examine how the memory allocation impacts the estimating accuracy. For the MAWI trace, we fix a total memory of 8 MB, vary the size of the S-sketch from 1 MB to 4 MB, and allocate the rest memory to the P-sketch; Similarly, with the FB trace, we fix a total memory of 1.75 MB, vary the size of the S-sketch from 4 kB to 768 kB, and allocate the rest to the P-sketch. We fix $b = 4$ for the HLL counters.

Fig. 8 presents the AREs achieved by PS-Sketch under different memory allocation schemes. We can see that with the MAWI trace, the minimum ARE is achieved under the configuration of (6 MB−4−2 MB), which means that the size of the P-sketch is 3 times of the S-sketch. On the other hand, with the FB trace, the minimum ARE is realized under (1.72 MB−4−32 kB), in which the P-sketch is 54 times larger than the S-sketch. Such a difference can be explained with different characteristics of the traffic traces. Recall that in the MAWI trace, a flow's persistent-spread is largely decided by its spread, so the S-sketch for spread estimating should be allocated as much as $\frac{1}{4}$ of the total memory. On the other hand, in the FB trace, a flow's persistent-spread is largely decided by the persistence of its belonging elements, so the P-sketch for persistence estimating should be allocated over 98% of the available memory for minimizing ARE.

*5) Discussion:*

*a) Analysis on errors:* The experiment results in Fig. 5-8 suggest that PS-Sketch is overall accurate in estimating network flows' persistent-spreads. However, for an individual flow, its persistent-spread could be either over-estimated or under-estimated. The over-estimating errors are caused by hash collisions in both the P-sketch and the S-sketch, and the errors impact all the network flows. On the other hand, as discussed in Sec. IV-B1, the under-estimating errors are caused by bit-shifting and HLL's small-range correction, which impact network flows with small spreads more severely. As a consequence, from Fig. 6 and 7, one can see that network flows with large persistent-spreads tend to be over-estimated, while flows with small persistent-spreads are more likely to be under-estimated. In particular, since many network flows in the FB trace have small spreads, their persistent-spread estimations are consistently under-estimated, as shown in Fig. 7.

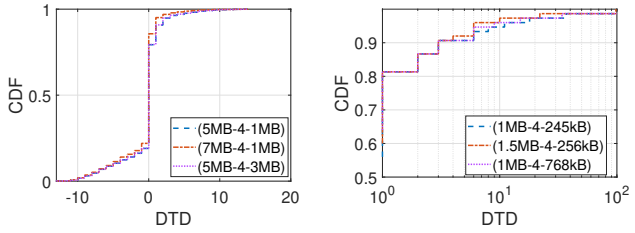
*b) How to configure?:* Our evaluation suggests that it is not always beneficial to employ more registers in the HLL counters due to the small-range correction of the HLL algorithm, and we suggest $b = 4$ for configuring the HLL counters. For the P-sketch and the S-sketch, our evaluation shows that it is always favorable to enlarge the two components for reducing the estimating errors, whose upper bound is given in Theorem 4. In particular, in (12), $\varepsilon_1$ and $\delta_1$ decide the P-sketch size of $w_1$ and $d_1$ as stated in Theorem 1, and $\varepsilon_2$ and $\delta_2$ decide $w_2$ and $d_2$, which is the size of the S-sketch as stated in Theorem 3. Given a memory budget and a target error bound, to configure PS-Sketch, one can choose $\varepsilon_1$, $\delta_1$, $\varepsilon_2$, and $\delta_2$ to ensure that the total memory usage does not exceed the budget, and under the configuration, the upper bound of the estimating error in (12) is below the target. One remaining issue is that the theoretical bound in (12) depends on the total persistence $P$, total spread $D$, and total persistent-spread $S$ of the elements and flows in the traffic, which is unknown in advance. Fortunately, as traffics on high-speed links are stable [31] and self-similar [32], these statistics can be obtained by analyzing pre-captured traffic traces in an offline way.

### C. Detecting Persistent-Spreaders

*1) Detecting accuracy:* In this section, we use PS-Sketch to detect persistent-spreaders. For the MAWI trace, we consider a flow as a persistent-spreader if its real persistent-spread as defined in (2) exceeds 200, and apply the threshold to detect the persistent-spreader flows in the S-sketch at the end of each epoch. For detecting persistent-spreaders in the FB trace, we use a threshold of 50.

In our evaluation for the MAWI trace, we start with the default configuration, enlarge the P-sketch from 5 MB to 7 MB, and enlarge the S-sketch from 1 MB to 3 MB respectively. Similarly, for the FB trace, we start with the default configuration, enlarge the P-sketch from 1 MB to 1.5 MB, and enlarge the S-sketch from 256 kB to 768 kB respectively.

Fig. 9 presents precisions and recalls of the detecting results. From Fig. 9(a), we can see that for the MAWI trace, PS-Sketch can achieve a precision around 70%, and a recall over 95% in detecting persistent-spreaders. Furthermore, enlarging P-sketch allows more elements to update the S-sketch, and consequently has more flows classified as persistent-spreaders, which explains the increased recall and decreased precision in Fig. 9(a). The observation suggests that enlarging the P-sketch




(a) MAWI (b) FB

Fig. 10. CDFs of DTDs for detecting persistent-spreaders (a) in MAWI trace under configurations of (5 MB−4−1 MB), (7 MB−4−1 MB), and (5 MB−4−3 MB), and (b) in FB trace under configurations of (1 MB−4−256 kB), (1.5 MB−4−256 kB), and (1 MB−4−768 kB).

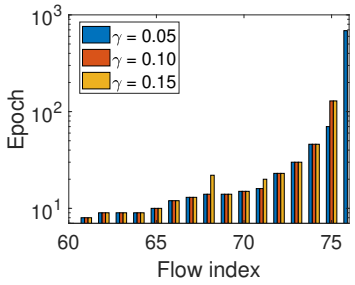



Fig. 11. Detecting time of flows by PS-Sketch with $\gamma = 0.05$, 0.10, and 0.15.

allows us to trade precision with recall, which is desirable as recall is of a higher importance than precision in detecting network anomalies.

Fig. 9(b) shows that when the P-sketch and S-sketch are sufficiently large, PS-Sketch realizes a precision of $100\%$ and a recall over $99.8\%$. Further investigation shows that only one real persistent-spreader flow is mis-classified. The result indicates that although using various approximation techniques including sketches and HLL, PS-Sketch can still achieve a very high detecting accuracy.

*2) Detecting timeliness:* To evaluate timeliness of the persistent-spreader detections, we compute DTD, which is the difference between the first epoch a flow is detected by PS-Sketch as a persistent-spreader and the first epoch that the flow's real persistent-spread exceeds the threshold (i.e., 200 for the MAWI trace and 50 for the FB trace). Note that a DTD could be 0, positive, or negative. A DTD of 0 means that the flow is detected in the same epoch that it becomes a real persistent-spreader, and a negative or positive DTD means that the detection is earlier or later.

In Fig. 10(a) and (b), we present CDFs of the DTDs of the detected persistent-spreaders under various configurations in the MAWI and the FB traces respectively. From both figures, we can see that the detections are timely. When detecting in the MAWI trace under the default configuration, as many as $60.31\%$ flows have a DTD of 0, and in the FB trace with the default configuration, $77.33\%$ of the flows have DTDs in $[-1, 1]$. In addition, by comparing the distributions, we can see that enlarging the P-sketch is slightly more helpful in reducing DTD than enlarging the S-sketch, as P-sketch is the first stage of the estimation, improving its accuracy benefits all the subsequent stages.

*3) Impact of element persistence decaying speed:* In (1), we define an element's persistence as the time-decaying sum of all its occurrences in the past, and the parameter $\gamma$ decides the decaying speed. In this section, we vary $\gamma$ from 0.05 to 0.15, and study how the parameter value impacts the persistent-spreader detections in the FB trace.

Fig. 11 presents detecting time of the persistent-spreaders detected by PS-Sketch with $\gamma = 0.05$, 0.10, and 0.15 respectively. From the figure we can see that with different $\gamma$ parameter values, PS-Sketch detects almost a same set of network flows in nearly the same epochs, and most of the flows are detected in early epochs of the trace. More specifically, 75 distinct flows are detected by PS-Sketch under all the three $\gamma$ parameter settings. With $\gamma = 0.05$, only one more flow, i.e., the flow "277cfd3e55f6a5da", is detected in the $687^{th}$ epoch. Moreover, 73 of 76 flows are detected in same epochs under different $\gamma$ values. When $\gamma$ is set as 0.05, the flow "277cfd3e55f6a5da" is detected 59 epochs earlier than the cases with $\gamma = 0.10$ and 0.15; and when $\gamma$ is set as 0.10, the flows "8a4655e48d34058d" and "0ea37886e741a715" are detected 8 and 4 epochs respectively earlier than the case with $\gamma = 0.15$.

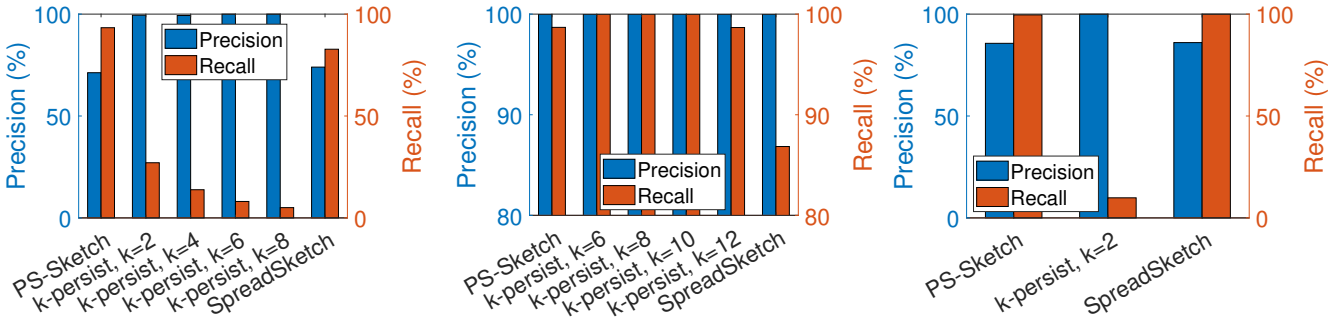
Fig. 11 suggests that the behaviors of the persistent-spreader flows in the trace are very different from the non persistent-spreaders, and can be consistently identified by PS-Sketch. With a smaller $\gamma$ value, PS-Sketch detects one more persistent-spreader, and detects a few persistent-spreader flows a little earlier. This is because with a slower element persistence decaying speed, elements have higher persistence values, and consequently larger persistent-spreads. Despite these differences, the element persistence decaying speed decided by $\gamma$ has a limited impact on the detecting results.

Overall, our experiment results suggest that PS-Sketch is capable to detect persistent-spreader flows in a timely manner accurately, and the detecting results are robust under various element persistence decaying speeds.

### D. Comparing with Existing Solutions

In this section, we compare PS-Sketch with two existing solutions. The first solution is based on computing network flows' $k$-persistent spreads [16]. As described in Sec. II, a flow's $k$-persistent spread is the number of the elements that occur in at least $k$ out of the $t$ recent epochs (i.e., $t$-occurrence$\geq k$), and a flow is defined as a $k$-persistent spreader if its $k$-persistent spread exceeds a pre-specified threshold. The second method under comparison is SpreadSketch [7], which detects super-spreader flows that have their estimated spreads greater than a threshold. Note that unlike PS-Sketch and the solution in [16], SpreadSketch [7] is designed for detecting super-spreaders rather than persistent-spreaders.

In our comparison, we apply the default configurations of PS-Sketch as described in Sec. IV-A for the MAWI, FB, and Witty traces respectively. We configure SpreadSketch into the same size of the S-sketch in PS-Sketch, but replacing each HLL counter with a multi-resolution bitmap [25]. When applying PS-Sketch and SpreadSketch, we set the thresholds as




(a) MAWI (b) FB (c) Witty

Fig. 12. Precisions and recalls for detecting persistent-spreaders by PS-Sketch, SpreadSketch, and method based on $k$-persistent spread (denoted as "$k$-persist") in (a) MAWI, (b) FB, and (c) Witty traces.

200 with the MAWI trace and 50 with the FB and Witty traces. When applying the method based on $k$-persistent spread, with the MAWI trace, we set $t = 8$, vary $k$ as 2, 4, 6, and 8, and set the threshold as $\frac{200}{k}$ so that the product of $k$ and the threshold is 200. Similarly, with the FB trace, we set $t = 16$, vary $k$ as 6, 8, 10, and 12, and set the threshold as $\frac{50}{k}$. With the Witty trace, we set $t = 8$, however, we only experiment with $k = 2$, as there is no $k$-persistent spreader in the trace when $k > 2$.

We apply the three methods to detect persistent-spreaders in the three traces, and present the results in terms of precisions and recalls in Fig. 12. From the figure, we can see that PS-Sketch has the overall best performance across the three traffic traces. In particular, with the MAWI trace that contains many super-spreaders but also a number of flows with highly persistent elements, as shown in Fig. 12(a), PS-Sketch achieves the highest recall by successfully detecting 93.33% of the real persistent-spreaders. On the other hand, with $k = 2$, only 27.06% of the persistent-spreader flows are detected as $k$-persistent spreaders, and the recall further decreases as $k$ is increased to 3, 4, and 5. This is reasonable as many flows in the trace are super-spreaders containing a large number of concurrent elements, but their elements have low persistence, thus can not be detected as $k$-persistent spreaders. SpreadSketch successfully detects the super-spreaders in the trace (i.e., network flows with their spread greater than the threshold in at least one epoch), but it fails to detect those flows that do not have large spreads but have their elements highly persistent, therefore has a lower recall of 82.77%.

For the FB trace in which all the persistent-spread flows contain highly persistent elements, Fig. 12(b) shows that the method based on $k$-persistent spread successfully detects all the 76 real persistent-spreaders until $k = 12$. This is reasonable as all the persistent-spreader flows in the trace are also $k$-persistent spreaders. PS-Sketch has a recall of 98.68% by missing only one persistent-spreader flow due to the underestimating error of the HLL counter. On the contrary, SpreadSketch has a much lower recall of 86.84%, as it misses the persistent-spreader flows whose spreads are not large enough but have highly persistent elements.

Finally, for the Witty trace that contains only super-spreaders, Fig. 12(c) shows that SpreadSketch, which is dedicatedly designed for detecting super-spreaders, successfully detects all the 264 real persistent-spreader flows. PS-Sketch miss-identifies one persistent-spreader due to hash collision in the S-sketch, and achieves a recall of 99.62%. On the other hand, with the method based on $k$-persistent spread, we have meaningful results only when $k = 2$, and with $k = 2$, the method only detects 26 flows, while misses the other 238 persistent-spreaders that have large spreads but few elements occurring in two or more epochs in $t = 8$ recent epochs.

In summary, our comparisons suggest that the existing solutions have their limitations: The method based on $k$-persistent spread [16] fails to detect the persistent-spreader flows that are super-spreaders in the MAWI and the Witty traces, and SpreadSketch [7] is unable to detect the persistent-spreaders containing highly-persistent elements in the MAWI and the FB traces. On the other hand, despite the diverse characteristics, PS-Sketch is capable to detect the persistent-spreader flows accurately across the traces all the time.

### *E. PS-Sketch on Programmable Switch*

We implement a PS-Sketch prototype with $P4_{16}$ [33] and compile it on an Edgecore Wedge 100BF-32X hardware programmable switch based on the Intel Barefoot Tofino chip [19]. We overcome the limitations of the programmable switch, such as the lack of support for float-point values, no support for multiplication, division, exponential and logarithmic operations, etc., with the following approaches.

First, in our definition of element persistent as in (1), when the value of $I_{m,i}$ is 1 and is decayed by a factor of $e^{-\gamma}$, its new value would be between 0 and 1, which is a float-point value not supported by the programmable switch. To overcome this limitation, in our implementation, $I_{m,i}$ is either 0 or 512, and when estimating an element's persistence as in (4), the raw value is right-shifted 9 bits to divide 512.

Second, to integrate an element's estimated persistence $\hat{P}_m$ to the HLL approximation, we right-shift the sub-string $H_2(m)$ $\lceil \log_2 \hat{P}_m \rceil$ bits. Unfortunately, logarithmic operation is not supported by the programmable switch. To solve this problem, we use a range-match table to perform the logarithmic computation. More specifically, we compare $\hat{P}_m$ with a range between $2^{i-1}$ and $(2^i - 1)$, and if $\hat{P}_m$ falls in such a range, we right-shift $H_2(m)$ $i$ bits to obtain $H_2'(m)$. Since the largest

TABLE II
SWITCH RESOURCE USAGE

| | Total | P-sketch | S-sketch |
|---|---|---|---|
| MAU Stages | 12 | 6 | 6 |
| Hash bit | 5.9% | 4.0% | 1.9% |
| ALU | 29.2% | 10.4% | 18.8% |
| SRAM | 11.7% | 9.5% | 2.2% |
| TCAM | 1.0% | 0.3% | 0.7% |
| VLIW instruction | 5.7% | 3.4% | 2.3% |
| PHV | 17.7% | - | - |

element persistence estimation is bounded by $\frac{e^\gamma}{e^\gamma-1}$, the table has a limited number of entries.

Finally, to compute $\rho(H_2'(m))$, i.e., position of the left-most '1' in the sub-string $H_2'(m)$, we use a longest-prefix-match (LPM) table to match the sub-strings to the patterns of $0^{(\rho-1)}1*$, and update the corresponding registers. The LPM table contains at most $L-b$ entries.

In the PS-Sketch prototype, we implement the P-sketch with $d_1 = 2$ rows and $w_1 = 2^{15}$ columns of buckets, and realize the S-sketch with $d_2 = 2$ rows and $w_2 = 2^{11}$ columns of buckets. The other parameters are set as in Sec. IV-A. Table II presents the consumptions of various resources on the programmable switch. From the table we can see that PS-Sketch consumes 12 stages, which is all the stages that the switch can provide, and among them, the P-sketch takes the first 6 stages and the S-sketch takes the remaining 6 stages. In addition to the overall resource usages, we also present resource usages of the P-sketch and the S-sketch in the table as well. We can see that PS-Sketch consumes relatively more ALU and SRAM, as the sketches, which are realized as register arrays, consume plenty of these resources. Anyway, PS-Sketch still leaves sufficient resources in each stage for other applications.

One potential issue of PS-Sketch is its stage usage, as the prototype uses up all the 12 stages. One possible modification is to split the P-sketch and the S-sketch and deploy them on two adjacent switches, and let the packets to carry $p.C$ and $p.F$ in their unused header fields such as VLAN or DSCP. In addition, the new generation of Tofino chip provides as many as 20 stages, which is sufficient for PS-Sketch. Finally, as an application successfully compiled and deployed on the Tofino switch, we find that the PS-Sketch prototype can process packets at the line-rate of the switch (i.e., $40\,\mathrm{Gbit/s}$).

## V. DISCUSSION

In this section, we discuss some alternative choices in designing PS-Sketch. In the S-sketch, we employ HLL as the distinct counter, and one alternative design choice is to use multi-resolution bitmap [25], as it is also updated by the leading '0's in the hash string of an element, and the hash string can also be right-shifted to encode the estimated element persistence. However, unlike HLL that uses registers, a multi-resolution bitmap employs $c$ bitmaps for the hash string patterns of $0^0 1*$, $0^1 1*$, $\cdots$, $0^{c-1}*$, and consumes more memory than HLL. For example, the multi-resolution bitmap in [7] is as large as 438 bits, which is over 5 times of the 80-bit HLL counter that we use in this work. Moreover, right-shifting hash strings produces more patterns, and require more bitmaps. Since P-sketch has removed duplicates by allowing only one packet of an element to reach to the S-sketch per epoch, another design choice is to simply use a counter instead of HLL. However, for tracking a persistent-spread up to $S_x$, a counter of $O(\log S_x)$ bits is needed, while HLL requires only $\Omega(\log\log S_x)$ bits [18]. For these reasons, we prefer HLL to multi-resolution bitmap or counter in our design.

As we have seen in Sec. IV, since an element updates the S-sketch only once, if a flow has a small spread, its persistent-spread tends to be under-estimated due to bit-shifting and HLL's small-range correction. One possible way to overcome this problem is to exploit the recirculation mechanism of the programmable switch [34], which allows the first packet of an element with an estimated persistence $\hat{P}_m$ to pass the pipeline and update the S-sketch $\lceil \hat{P}_m \rfloor$ times. Note that to have randomized hash strings, each time the packet passes the pipeline, it has its element ID changed as $m' = (x, H(m))$, where $m$ is its element ID in the previous pipeline pass. However, recirculating such a large number of packets for so many times will consume considerable processing and memory resources within the switch, and impose substantial complexity to the system design. For example, it is challenging to ensure that all the recirculations end within the right epoch. On the other hand, as we have seen in Sec. IV, most of the flows heavily under-estimated by the HLL algorithm are the ones with small spreads, which are unlikely to be persistent-spreaders. For this reason, we prefer bit-shifting to packet recirculating in our design.

## VI. RELATED WORK

### A. Spread and Persistence Estimating

To detect network flows with large spreads, Snort [35] and FlowScan [36] maintain all active connections for each source address, however, per-source tracking is not scalable in high-speed networks. To reduce the overhead, Cao et al. [37] propose to filter out network flows with small cardinality, and employ a thresholded bitmap to detect super-spreaders. Li et al. [4] present a dynamic bit-sharing technique to track network flows, and apply the maximum-likelihood estimation to detect heavy spreaders. Huang et al. [5] develop an on-chip method to sample packets without duplications, and estimate per-flow spread with the sampled packets in the off-chip memory. Zhang et al. [6] propose a memory-efficient method for estimating spreads of network flows under any arbitrary flow definition. Su et al. [38] extend the virtual bitmap estimator [9] to estimate spreads for either 1D or 2D hierarchical flows. SpreadSketch [7] extends the Count-Min sketch [20] by replacing the conventional counter in each sketch bucket with a multi-resolution bitmap [25]. Despite these efforts, one major drawback of spreader detection is that an attacker can bypass the detection if it reduces its spread but attacks more persistently [9].

There is a rich literature on estimating element persistence in data streams. SS [10] uses a hash-based filter to sample elements in each epoch. PIE [11] stores Raptor codes [39] in a Space-Time Bloom filter (STBF), and decodes ID of a persistent element if sufficient Raptor codes are found in

STBF. Chen et al. [12] apply Bloom filter and load-balancing techniques to reduce memory overhead for recording persistent elements. On-Off sketch [13] extends the Count-Min sketch [20] by associating the counter in each sketch bucket with an On/Off state, which prevents the sketch from over-estimating an item's persistence when it repeatedly appears in one epoch. P-Sketch [14] extends the On-Off sketch by tracking arrival continuities of the items, and uses a probabilistic mechanism to evict elements on hash collisions. The authors further refine bucket replacement decisions with multi-dimensional information [40]. However, these works focus on individual elements rather than attackers, and are based on software.

### B. Persistent-Spread Estimating

To estimate persistent-spreads of network flows that appear in all the $t$ recent epochs. Xiao et al. [9] propose a data structure called multi-virtual bitmaps to estimate persistent-spreads of flows in high-speed networks. Zhou et al. [15] present a data structure named Virtual Intersection HyperLogLog (VI-HLL) that is more memory efficient. Huang et al. [16] generalize the problem by estimating number of the elements in a flow that appear in no less than $k$ out of the $t$ recent epochs. However, as we have discussed in Sec. II, under the conventional definition of persistent-spreader as in these works, an attacker can bypass the detection by manipulating its contact pattern and making a tradeoff between persistence and spread. Moreover, existing works adopt an online/offline hybrid method, where the complicated operations for constructing the estimators are conducted by software. A recent work [41] proposes a sketch named WavingSketch that is versatile to track top-$K$ persistent elements or super-spreaders, however, the tasks can not be conducted simultaneously.

Our work differs from existing works in two aspects. First, our new definitions on persistence and persistent-spread are more accurate, and are difficult for an attacker to bypass. Second, our solution is compatible with hardware programmable switches, and processes packets at line-rate of the switch.

### C. Sketch for Network Traffic Monitoring

Sketch-based methods are widely used to track traffic characteristics in high-speed networks. To collect per-flow statistics, Elastic Sketch [42] employs a hash table and a Count-Min sketch [20] to monitor elephant and mice flows respectively; Gu et al. [43] propose a mechanism that allows switches to collaborate in network-wide per-flow measurement tasks; $\mu$Mon [44] presents WaveSketch to perform wavelet transforms with switch hardware for detecting microsecond-scale flow events. To detect heavy-hitter flows, HashPipe [45] uses a pipeline of hash tables to retain heavy flows while evicting lighter flows over time. To detect packet losses, LossRadar [46] presents a Invertible Bloom Filter (IBF) to capture individual lost packet and report the location where the losses happen; ChameleMon [47] proposes a novel sketch named FermatSketch, and automatically allocates on-chip memory between packet accumulation and loss detection tasks.

### D. Programmable Packet Processing in Hardware

To deploy sketch-based data structures and algorithms on hardware programmable switch. UnivMon [48] develops a framework for flow monitoring, and provides generality and high accuracy with programmable switches. Sketchovsky [49] presents a cross-sketch optimization and composition framework to ensemble sketch instances for diverse measurement tasks on programmable switches. HeteroSketch [50] provides a network-wide flow monitoring framework that coordinates sketch-based measurement to determine task placement and resource allocation for a network of heterogeneous devices.

For other packet processing tasks, MacDavid et al. [51] exploit low-pass filter (LPF) in Tofino switch for fairly allocating bandwidth among network flows. Chen et al. [22] realize exponential weighted moving average (EWMA) with P4 on hardware programmable switches to cope with the burstiness in TCP traffic for weighted fair queueing (WFQ) [52].

## VII. Conclusion

Detecting persistent-spreaders is essential for preventing attacks in high-speed networks. In this paper, we present PS-Sketch, a novel system for estimating persistent-spreads of network flows and detecting persistent-spreaders in network data streams. The design of PS-Sketch is centered around our formal definitions on element persistence and network flow's persistent-spread. We define an element's persistence as the time-decaying sum of all its occurrences in the past, which better captures the element's behavior than counting its occurrences in the $t$ recent epochs as in the conventional definition. Based on element persistence, we define a network flow's persistent-spread as the sum of the persistence of the elements that belong to the flow, and consider a flow as a persistent-spreader if its persistent-spread exceeds a threshold. Our definition of persistent-spreader covers a wide range of network flows from super-spreaders to stealthy DDoS attackers, and is difficult for attackers to bypass.

PS-Sketch employs a sketch-based approach to estimate network flows' persistent-spreads with two adjacent sketches, namely the P-sketch and the S-sketch. In the P-sketch, we exploit the feature of low-pass filter (LPF) in programmable switch to compute the time-decaying element persistence, and in the S-sketch, we employ the HyperLogLog (HLL) algorithm to estimate network flows' persistent-spreads. In particular, we extend the HLL algorithm to integrate an element's estimated persistence to the spread estimating of the flow that the element belongs to. Theoretical analysis and trace-driven evaluation show that PS-Sketch achieves high accuracy in estimating persistent-spreads for network flows, detects persistent-spreaders in a timely manner, and outperform the existing solution. We further implement PS-Sketch in P4 and demonstrate that it is feasible to be deployed on commodity hardware switches.

## Appendix A
## Proofs

### A. Proof of Theorem 1

**Theorem 1.** *For $w_1 = \lceil e/\varepsilon_1 \rceil$ and $d_1 = \lceil \ln(1/\delta_1) \rceil$, with a probability at least $1 - \delta_1$, $\hat{P}_m$ is no larger than $P_m + \varepsilon_1 P$,*

*where $P = \sum_m P_m$. In other words, we have*

$$\Pr[\hat{P}_m \le P_m + \varepsilon_1 P] \ge 1 - \delta_1 \tag{5}$$

*Proof.* Let $\mathbb{M}_k$ be the set of the elements that occur in an epoch $t_k \in \{\cdots, t_{-1}, t_0\}$, and at the current epoch $t_0$, the value added by the elements in $\mathbb{M}_k$ is decayed by a factor of $e^{-k\gamma}$. By the definition in (1), we have

$$\sum_{t_k \in \{\cdots, t_{-1}, t_0\}} |\mathbb{M}_k| e^{-k\gamma} = \sum_m P_m = P \tag{14}$$

Consider the LPF $\mathbf{B}_i[h_i(m)].C$ of an element $m$, let $\mathbb{T}_m$ be the set of the epochs that the element $m$ occurs, and let $\overline{\mathbb{T}}_m = \{\cdots, t_{-1}, t_0\} - \mathbb{T}_m$ be the set of the epochs that $m$ does not occur. In an epoch $t_k \in \overline{\mathbb{T}}_m$, an element $m' \in \mathbb{M}_k$ is hashed to $\mathbf{B}_i[h_i(m)]$ (i.e., $h'_i(m) = h_i(m)$, but $m' \neq m$) with a probability of $\frac{1}{w_1}$, so during the epoch, the probability that $\mathbf{B}_i[h_i(m)].C$ is mistakenly increased is $1 - (1 - \frac{1}{w_1})^{|\mathbb{M}_k|} \approx \frac{|\mathbb{M}_k|}{w_1}$, and at the current epoch $t_0$, the mistakenly increased value is decayed by a factor of $e^{-k\gamma}$. Let $\Delta_i P_m = \mathbf{B}_i[h_i(m)].|C| - P_m$, which is the value that $\mathbf{B}_i[h_i(m)].C$ is mistakenly increased by the elements other than $m$ due to hash collisions, then according to (14), for all the epochs in $\overline{\mathbb{T}}_m$, we have

$$\begin{aligned} E[\Delta_i P_m] &= \sum_{t_k \in \overline{\mathbb{T}}_m} \frac{1 - (1 - \frac{1}{w_1})^{|\mathbb{M}_k|}}{e^{k\gamma}} \\ &\le \frac{\sum_{t_k \in \{\cdots, t_{-1}, t_0\}} |\mathbb{M}_k|}{w_1 e^{k\gamma}} \\ &= \frac{P}{w_1} \le \frac{\varepsilon_1 P}{e} \end{aligned} \tag{15}$$

Since $\Pr[\hat{P}_m - P_m \le \varepsilon_1 P] = 1 - \Pr[\hat{P}_m - P_m > \varepsilon_1 P]$, and $\Pr[\hat{P}_m - P_m > \varepsilon_1 P] = \Pr[\forall i, \Delta_i P_m > \varepsilon_1 P] = (\Pr[\Delta_i P_m > \varepsilon_1 P])^{d_1}$, from (15) and according to Markov's inequity [53],

$$\Pr[\Delta_i P_m > \varepsilon_1 P] \le \frac{E[\Delta_i P_m]}{\varepsilon_1 P} \le \frac{1}{e} \tag{16}$$

So, $\Pr[\hat{P}_m - P_m > \varepsilon_1 P] \le e^{-d_1} \le \delta_1$, which means that with a probability at least $1 - \delta_1$, we have $\hat{P}_m \le P_m + \varepsilon_1 P$. □

### B. Proof of Theorem 2

**Theorem 2.** *For a network flow $x$, $\tilde{S}_x \ge (1-\sigma) S_x$, and with a probability at least $1 - \delta_1$, we have*

$$\tilde{S}_x \le 2(1+\sigma)(S_x + \varepsilon_1 P D_x) \tag{9}$$

*where $D_x = \sum_{m \in x} I_{m,0}$ is flow $x$'s spread in epoch $t_0$.*

*Proof.* Let $\bar{S}_x = \sum_{m \in x} 2^{\lceil \log \hat{P}_m \rceil} \cdot I_{m,0}$. Since $\hat{P}_m \le 2^{\lceil \log \hat{P}_m \rceil} \le 2\hat{P}_m$, it is easy to see that

$$\sum_{m \in x} \hat{P}_m \cdot I_{m,0} \le \bar{S}_x \le 2 \sum_{m \in x} \hat{P}_m \cdot I_{m,0} \tag{17}$$

As the P-sketch only over-estimates by having $\hat{P}_m \ge P_m$, we have

$$\bar{S}_x \ge \sum_{m \in x} \hat{P}_m \cdot I_{m,0} \ge \sum_{m \in x} P_m \cdot I_{m,0} = S_x \tag{18}$$

On the other hand, according to Theorem 1, with a probability at least $1 - \delta_1$, we have

$$\begin{aligned} \bar{S}_x &\le 2 \sum_{m \in x} \hat{P}_m \cdot I_{m,0} \\ &\le 2 \sum_{m \in x} (P_m + \varepsilon_1 P) \cdot I_{m,0} \le 2(S_x + \varepsilon_1 P D_x) \end{aligned} \tag{19}$$

where $D_x = \sum_{m \in x} I_{m,0}$ is flow $x$'s spread in epoch $t_0$.

If the flow $x$ is estimated by one dedicated HLL, then $\bar{S}_x$ is the ground-truth of the HLL approximation, and $\bar{S}_x \cdot (1-\sigma) \le \tilde{S}_x \le \bar{S}_x \cdot (1+\sigma)$, where $\sigma$ is multiples of $\frac{1.04}{\sqrt{s}}$ [18].

From (18), we have

$$\tilde{S}_x \ge (1-\sigma)\bar{S}_x \ge (1-\sigma) S_x \tag{20}$$

and from (19), one can see that with a probability at least $1 - \delta_1$,

$$\tilde{S}_x \le (1+\sigma)\bar{S}_x \le 2(1+\sigma)(S_x + \varepsilon_1 P D_x) \tag{21}$$

□

### C. Proof of Theorem 3

**Theorem 3.** *For $w_2 = \lceil e/\varepsilon_2 \rceil$ and $d_2 = \lceil \ln(1/\delta_2) \rceil$, with a probability at least $1 - \delta_2$, $\hat{S}_x$ is no larger than $\tilde{S}_x + \varepsilon_2 \tilde{S}$, where $\tilde{S} = \sum_x \tilde{S}_x$. In other words, we have*

$$\Pr[\hat{S}_x \le \tilde{S}_x + \varepsilon_2 \tilde{S}] \ge 1 - \delta_2 \tag{11}$$

*Proof.* Consider an HLL $\mathbf{C}_i[g_i(m)].H$ of a flow $x$, and let $\Delta_i \tilde{S}_x = \mathbf{C}_i[g_i(m)].|H| - \tilde{S}_x$ be the value introduced to $\mathbf{C}_i[g_i(m)].H$ by the flows other than $x$ due to hash collisions. Since a flow $x' \neq x$ has $g_i(x') = g_i(x)$ at a probability of $\frac{1}{w_2}$, let $\tilde{S}_{-x} = \sum_{x' \neq x} \tilde{S}_{x'}$, then the amount that $\mathbf{C}_i[g_i(m)].H$ is mistakenly increased is $1 - (1 - \frac{1}{w_2})^{\tilde{S}_{-x}} \approx \frac{\tilde{S}_{-x}}{w_2}$. It is easy to see that

$$E[\Delta_i \tilde{S}_x] = \frac{\tilde{S}_{-x}}{w_2} \le \frac{\tilde{S}}{w_2} \le \frac{\varepsilon_2 \tilde{S}}{e} \tag{22}$$

Since $\Pr[\hat{S}_x - \tilde{S}_x \le \varepsilon_2 \tilde{S}] = 1 - \Pr[\hat{S}_x - \tilde{S}_x > \varepsilon_2 \tilde{S}]$, and $\Pr[\hat{S}_x - \tilde{S}_x > \varepsilon_2 \tilde{S}] = \Pr[\forall i, \Delta_i \tilde{S}_x > \varepsilon_2 \tilde{S}] = (\Pr[\Delta_i \tilde{S}_x > \varepsilon_2 \tilde{S}])^{d_2}$, according to Markov's inequity [53],

$$\Pr[\Delta_i \tilde{S}_x > \varepsilon_2 \tilde{S}] \le \frac{E[\Delta_i \tilde{S}_x]}{\varepsilon_2 \tilde{S}} \le \frac{1}{e} \tag{23}$$

So, $\Pr[\hat{S}_x - \tilde{S}_x > \varepsilon_2 \tilde{S}] \le e^{-d_2} \le \delta_2$, which means that with a probability at least $1 - \delta_2$, we have $\hat{S}_x \le \tilde{S}_x + \varepsilon_2 \tilde{S}$. □

### D. Proof of Theorem 4

**Theorem 4.** *Given $\varepsilon_1$, $\varepsilon_2$, $\delta_1$, $\delta_2$, and $\sigma$, with a probability at least $(1 - \delta_1)(1 - \delta_2)$, we have*

$$\begin{aligned} \hat{S}_x \le\ & 2(1+\sigma) S_x + 2\varepsilon_1 (1+\sigma) P D_x \\ & + 2\varepsilon_2 (1+\sigma) S + 2\varepsilon_1 \varepsilon_2 (1+\sigma) P D \end{aligned} \tag{12}$$

*where $P = \sum_m P_m$, $S = \sum_x S_x$, $D = \sum_x D_x$, and $D_x = \sum_{m \in x} I_{m,0}$ is flow $x$'s spread in epoch $t_0$.*

*Proof.* According to Theorem 2, with a probability at least $1-\delta_1$, $\tilde{S}_x \leq 2(1+\sigma)(S_x+\varepsilon_1 PD_x)$, we can see that with the same probability,

$$\begin{aligned}\tilde{S} = \sum_x \tilde{S}_x &\leq 2(1+\sigma)\sum_x (S_x+\varepsilon_1 PD_x) \qquad (24)\\ &= 2(1+\sigma)(S+\varepsilon_1 PD)\end{aligned}$$

From (9) and (24), and according to Theorem 3, with a probability at least $(1-\delta_1)\cdot(1-\delta_2)$, we have

$$\begin{aligned}\hat{S}_x &\leq \tilde{S}_x + \varepsilon_2 \tilde{S} \qquad (25)\\ &\leq 2(1+\sigma)(S_x+\varepsilon_1 PD_x) + 2\varepsilon_2(1+\sigma)(S+\varepsilon_1 PD)\\ &= 2(1+\sigma)S_x + 2\varepsilon_1(1+\sigma)PD_x\\ &\quad +2\varepsilon_2(1+\sigma)S + 2\varepsilon_1\varepsilon_2(1+\sigma)PD\end{aligned}$$

□

## REFERENCES


[1] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic DDoS defense," in *Proc. USENIX Security'15*, Washington, D.C., USA, Aug. 2015, pp. 817–832.

[2] Z. Durumeric, M. Bailey, and J. A. Halderman, "An Internet-wide view of Internet-wide scanning," in *Proc. USENIX Security'14*, San Diego, CA, USA, Aug. 2014, pp. 65–78.

[3] C. Shannon and D. Moore, "The spread of the Witty worm," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 46–50, 2004.

[4] T. Li, S. Chen, W. Luo, M. Zhang, and Y. Qiao, "Spreader classification based on optimal dynamic bit sharing," *IEEE/ACM Trans. Networking*, vol. 21, no. 3, pp. 817–830, 2013.

[5] H. Huang, Y.-E. Sun, C. Ma, S. Chen, Y. Du, H. Wang, and Q. Xiao, "Spread estimation with non-duplicate sampling in high-speed networks," *IEEE/ACM Trans. Networking*, vol. 29, no. 5, pp. 2073–2086, 2021.

[6] H. Zhang, H. Huang, Y.-E. Sun, and Z. Wang, "MIME: Fast and accurate flow information compression for multi-spread estimation," in *Proc. IEEE ICNP'23*, Reykjavik, Iceland, Oct. 2023, pp. 1–11.

[7] L. Tang, Y. Xiao, Q. Huang, and P. P. C. Lee, "A high-performance invertible sketch for network-wide superspreader detection," *IEEE/ACM Trans. Networking*, vol. 31, no. 2, pp. 724–737, 2023.

[8] F. Giroire, J. Chandrashekar, N. Taft, E. Schooler, and D. Papagiannaki, "Exploiting temporal persistence to detect covert botnet channels," in *Proc. International Symposium on Recent Advances in Intrusion Detection, LNCS 5758*, Saint-Malo, France, Sep. 2009, pp. 326–345.

[9] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen, "Estimating the persistent spreads in high-speed networks," in *Proc. IEEE ICNP'14*, Raleigh, NC, USA, Dec. 2014, pp. 131–142.

[10] B. Lahiri, J. Chandrashekar, and S. Tirthapura, "Space-efficient tracking of persistent items in a massive data stream," in *Proc. ACM International Conference on Distributed Event-Based Systems (DEBS'11)*, New York, NY, USA, Jul. 2011, pp. 255–266.

[11] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, "Finding persistent items in data streams," *Proc. VLDB Endowment*, vol. 10, no. 4, pp. 289–300, 2016.

[12] L. Chen, H. Dai, L. Meng, and J. Yu, "Finding needles in a hay stream: On persistent item lookup in data streams," *Computer Networks*, vol. 181, pp. 107 518, 1–11, 2020.

[13] Y. Zhang, J. Li, Y. Lei, T. Yang, Z. Li, G. Zhang, and B. Cui, "On-off sketch: a fast and accurate sketch on persistence," *Proc. VLDB Endowment*, vol. 14, no. 2, pp. 128–140, 2020.

[14] W. Li and P. Patras, "P-Sketch: A fast and accurate sketch for persistent item lookup," *IEEE/ACM Trans. Networking*, vol. 32, no. 2, pp. 987–1002, 2024.

[15] Y. Zhou, Y. Zhou, M. Chen, and S. Chen, "Persistent spread measurement for big network data based on register intersection," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, 2017, article no. 15.

[16] H. Huang, Y.-E. Sun, C. Ma, S. Chen, Y. Zhou, W. Yang, S. Tang, H. Xu, and Y. Qiao, "An efficient $k$-persistent spread estimator for traffic measurement in high-speed networks," *IEEE/ACM Trans. Networking*, vol. 28, no. 4, pp. 1463–1476, 2020.

[17] "P4-tofino-examples/tna_meter_lpf_wred," accessed on Sep. 14, 2024. [Online]. Available: https://github.com/zhaoyboo/p4-tofino-examples/tree/main/tna_meter_lpf_wred/tna_meter_lpf_wred.p4

[18] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. Conference on Analysis of Algorithms (AofA'07)*, Juan des Pins, France, Jun. 2007, pp. 137–156.

[19] "Intel Tofino series," accessed on Mar. 22, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html

[20] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. of Algorithms*, vol. 55, no. 1, 2005.

[21] "How to get queue size periodically to calculate any statistical parameter like average,mean,median,variance?" accessed on Sep. 14, 2024. [Online]. Available: https://forum.p4.org/t/how-to-get-queue-size-periodically-to-calculate-any-statistical-parameter-like-average-mean-median-variance

[22] W. Chen, Y. Tian, X. Yu, B. Zheng, and X. Zhang, "Enhancing fairness for approximate weighted fair queueing with a single queue," *IEEE/ACM Trans. Networking*, vol. 32, no. 5, pp. 3901–3915, 2024.

[23] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Computer and System Sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[24] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Proc. European Symposium on Algorithms (ESA'03), LNCS 2832*, Budapest, Hungary, Sep. 2003, pp. 605–617.

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

[26] "MAWI working group traffic archive," accessed on Mar. 22, 2024. [Online]. Available: https://mawi.wide.ad.jp/mawi/

[27] "DC traces," accessed on Mar. 22, 2024. [Online]. Available: https://trace-collection.net/dc-traces/

[28] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM SIGCOMM'15*, London, UK, Aug. 2015, pp. 123–137.

[29] "The UCSD network telescope," accessed on Mar. 10, 2025. [Online]. Available: https://www.caida.org/projects/network_telescope/

[30] "Witty worm dataset," accessed on Mar. 10, 2025. [Online]. Available: https://www.caida.org/catalog/datasets/witty_worm_dataset/

[31] P. Velan, J. Medkov, T. Jirsík, and P. Čeleda, "Network traffic characterisation using flow-based statistics," in *Proc. NOMS'16*, Istanbul, Turkey, Apr. 2016, pp. 907–912.

[32] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Trans. Networking*, vol. 2, no. 1, pp. 1–15, 2002.

[33] "$P4_{16}$ portable switch architecture (PSA)," The P4.org Architecture Working Group, Tech. Rep., 2021.

[34] "How to use recirculation on tofino?" accessed on Sep. 14, 2024. [Online]. Available: https://forum.p4.org/t/how-to-use-recirculation-on-tofino

[35] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. USENIX LISA'99*, Seattle, WA, USA, Nov. 1999, pp. 229–238.

[36] D. Plonka, "FlowScan: A network traffic flow reporting and visualization tool," in *Proc. USENIX LISA'00*, New Orleans, LA, USA, Dec. 2000, pp. 305–317.

[37] J. Cao, Y. Jin, A. Chen, T. Bu, and Z.-L. Zhang, "Identifying high cardinality Internet hosts," in *Proc. IEEE INFOCOM'09*, Rio de Janeiro, Brazil, Apr. 2009, pp. 810–818.

[38] H. Su and Q. Xiao, "Online detection of 1d and 2d hierarchical superspreaders in high-speed networks," in *Proc. Asia-Pacific Workshop on Networking (APNet'23)*, Hong Kong, China, Jun. 2023, pp. 109–115.

[39] A. Shokrollahi, "Raptor codes," *IEEE Trans. Information Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[40] W. Li and P. Patras, "Stable-Sketch: A versatile sketch for accurate, fast, web-scale data stream processing," in *Proc. WWW'24*, Singapore, May 2024, pp. 4227–4238.

[41] J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. Zhang, "WavingSketch: An unbiased and generic sketch for finding top-$k$ items in data streams," in *Proc. KDD'20*, Virtul Event, CA, USA, Aug. 2020, pp. 1574–1584.

[42] 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. ACM SIGCOMM'18*, Budapest, Hungary, Aug. 2018, pp. 561–575.

[43] L. Gu, Y. Tian, W. Chen, Z. Wei, C. Wang, and X. Zhang, "Per-flow network measurement with distributed sketch," *IEEE/ACM Trans. Networking*, vol. 32, no. 1, pp. 411–426, 2024.

[44] H. Zheng, C. Huang, X. Han, J. Zheng, X. Wang, C. Tian, W. Dou, and G. Chen, "$\mu$Mon: Empowering microsecond-level network monitoring with wavelets," in *Proc. ACM SIGCOMM'24*, Sydney, Australia, Aug. 2024, pp. 274–290.

[45] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. SOSR'17*, Santa Clara, CA, USA, Apr. 2017, pp. 164–176.

[46] Y. Li, R. Miao, C. Kim, and M. Yu, "LossRadar: Fast detection of lost packets in data center networks," in *Proc. of CoNEXT'16*, Irvine, CA, USA, Dec. 2016, pp. 481–495.

[47] K. Yang, Y. Wu, R. Miao, T. Yang, Z. Liu, Z. Xu, R. Qiu, Y. Zhao, H. Lv, Z. Ji, and G. Xie, "ChameleMon: Shifting measurement attention as network state changes," in *Proc. ACM SIGCOMM'23*, New York, NY, USA, Aug. 2023, pp. 881–903.

[48] 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'16*, Florianopolis, Brazil, Aug. 2016, pp. 101–114.

[49] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "Sketchovsky: Enabling ensembles of sketches on programmable switches," in *Proc. USENIX NSDI'23*, Boston, MA, USA, Apr. 2023, pp. 1273–1292.

[50] A. Agarwal, Z. Liu, and S. Seshan, "Heterosketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks," in *Proc. USENIX NSDI'24*, Renton, WA, USA, Apr. 2024, pp. 719–741.

[51] R. MacDavid, X. Chen, and J. Rexford, "Scalable real-time bandwidth fairness in switches," *IEEE/ACM Trans. Networking*, vol. 32, no. 2, pp. 1423–1434, 2024.

[52] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Trans. Networking*, vol. 1, no. 3, pp. 344–357, 1993.

[53] E. M. Stein and R. Shakarchi, *Real Analysis: Measure Theory, Integration, and Hilbert Spaces*. Princeton, NJ, USA: Princeton University Press, 2005.

**Zhaohui Wang** received the bachelor's degree in computer science from University of Electronic Science and Technology of China, Chengdu, China, in 2023. She is currently pursuing the master's degree with the School of Computer Science and Technology, University of Science and Technology of China. Her research interest is focused on network security.

**Ye Tian** received the bachelor's degree in electronic engineering and the master's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2001 and 2004, respectively, and the Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China, in 2007. He joined USTC in 2008 and is currently an Associate Professor with the School of Computer Science and Technology, USTC. His research interests include programmable networks, network measurement, and network security. He has published over 90 papers and co-authored a research monograph published by Springer. He is the winner of the Wilkes Best Paper Award of Oxford The Computer Journal in 2016. He is a member of the IEEE.

**Yiwen Wu** received the bachelor's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2023. He is currently pursuing the master's degree with the School of Computer Science and Technology, USTC. His research interest is focused on network security.

**Wei Chen** received the bachelor's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2020. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, USTC. His research interests include programmable networks and network measurement.

**Xinyu Zhang** received the bachelor's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2024. He is currently pursuing the master's degree with the School of Computer Science and Technology, USTC. His research interest is focused on programmable networks.

**Xinming Zhang** received the BE and ME degrees in electrical engineering from China University of Mining and Technology, Xuzhou, China, in 1985 and 1988, respectively, and the PhD degree in computer science and technology from the University of Science and Technology of China (USTC), Hefei, China, in 2001. Since 2002, he has been with the faculty of USTC, where he is currently a Professor with the School of Computer Science and Technology. From September 2005 to August 2006, he was a visiting Professor with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea. His research interest includes wireless networks, deep learning, and intelligent transportation. He has published more than 100 papers. He won the second prize of Science and Technology Award of Anhui Province of China in Natural Sciences in 2017. He won the awards of Top reviewers (1%) in Computer Science & Cross Field by Publons in 2019. He is a senior member of the IEEE.