PISketch: Finding Persistent and Infrequent Flows

Zhuochen Fan, Zhoujing Hu, Yuhan Wu, Jiarui Guo, Sha Wang, Wenrui Liu, Tong Yang, *Member, IEEE*, Yaofeng Tu, and Steve Uhlig

***Abstract*— Finding persistent and low-active activity periods is very helpful in practice, for example to detect intrusion activities. Most of the literature focuses on finding persistent flows or frequent flows. No previous work is able to find persistent and infrequent flows. In this paper, we propose a novel sketch data structure, PISketch, to find persistent and infrequent flows in real time. The key idea of PISketch is to define a weight and its Reward and Penalty System for each flow to combine and balance the information of both persistency and infrequency, and to keep high-weighted flows in a limited space through a strategy. We implement PISketch on P4, FPGA, and CPU platforms, and compare the performance of PISketch with two strawman solutions (On-Off + CM sketch, and PIE + CM sketch), in terms of finding persistent and infrequent flows. Our experimental results demonstrate the advantage of PISketch, by comparing it to two strawman solutions: 1) The F1 Score of PISketch is around 22.1% and 57.6% higher than two strawman solutions, respectively; 2) The Average Relative Error (ARE) of PISketch is around 820.9 (up to 1188.8) and 126.2 (up to 265.6) times lower than two strawman solutions, respectively; 3) The insertion throughput of PISketch is around 1.23 and 16.5 times higher than two strawman solutions, respectively. Moreover, we implement two concrete cases of PISketch through end-to-end experiments. All of our codes are available at GitHub.**

***Index Terms*— Data streams, persistent flows, infrequent flows, advanced persistent threats, sketch, weight, P4, FPGA.**

Manuscript received 19 July 2022; revised 17 December 2022; accepted 25 April 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Park. Date of publication 11 May 2023; date of current version 19 December 2023. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101390001 and in part by the National Natural Science Foundation of China (NSFC) under Grant U20A20179. The preliminary version of this paper titled "PISketch: Finding Persistent and Infrequent Flows" [DOI: 10.1145/3528082.3544834] was published in the Proceedings of the ACM SIGCOMM Workshop on Formal Foundations and Security of Programmable Network Infrastructures, August 22 (https://doi.org/10.1145/3528082.3544834). *(Zhuochen Fan, Zhoujing Hu, and Yuhan Wu are co-primary authors.) (Corresponding author: Tong Yang.)*

Zhuochen Fan, Zhoujing Hu, Yuhan Wu, Jiarui Guo, and Wenrui Liu are with the National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China (e-mail: fanzc@pku.edu.cn; rwy@pku.edu.cn; yuhan.wu@pku.edu.cn; ntguojiarui@pku.edu.cn; liuwenrui@pku.edu.cn).

Sha Wang is with the College of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: ws0623zz@163.com).

Tong Yang is with the National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China, and also with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: yangtongemail@gmail.com).

Yaofeng Tu is with the ZTE Nanjing Research and Development Center, Nanjing 210012, China (e-mail: tu.yaofeng@zte.com.cn).

Steve Uhlig is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, U.K. (e-mail: steve.uhlig@qumul.ac.uk).

Digital Object Identifier 10.1109/TNET.2023.3272287

I. INTRODUCTION

A. Background and Motivation

FINDING frequent flows and persistent flows have been considered as two important tasks in approximate data stream processing and network measurement [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. Here, we define the persistency of flow f as the number of time windows where f occurs [2], while time-based windows mean that the window size is defined as a fixed period of time. Differently, we find that finding persistent and infrequent flows in data streams is also important, for example to identify activities that are sustained but low-active. Note that a flow in this paper is defined as a part of the five tuples: source IP address, destination IP address, source port, destination port, and protocol. Let us describe three possible use cases for finding persistent and infrequent flows.

- **Case 1.** Attack defense. Many cyber attacks like Advanced Persistent Threats (APT) [14], [15] prefer persistently and covertly intruding target streaming databases to evade detection.
- **Case 2.** High-risk service discovery. In enterprise networks, high-risk services like Fast Reverse Proxy (FRP) [16] can expose local servers behind a Network Address Translation (NAT) or firewall to the Internet. It supports HTTP/HTTPS, TCP, UDP and many other protocols, and forwards requests to internal services through domain names. These connections are characterized by persistence and infrequency: FRP persistently produces packets, but the number of produced packets is very limited.
- **Case 3.** Lightweight heartbeat packet detection. A heartbeat is a periodic packet that keeps the persistent connection alive ($e.g.$, TCP keep-alive) and synchronizes the state, which is always persistent and infrequent. By detecting heartbeat packets, network administrators can count the number of persistent connections [17], [18]. However, traditional measurement solutions are not enough to support this. They often use a timeout mechanism to measure the number of active flows in the recent time window [19]. Therefore, a more convenient way to detect persistent and infrequent flows spanning a large number of windows is needed. Moreover, each persistent connection is a customer client in the financial market network. Once a large amount of critical information needs to be quickly distributed and transmitted to each client, the network may experience bursts and

sudden congestion. For network administrators, knowing the number of persistent connections allows them to estimate potential bursts and prevent them [20].

Finding persistent and infrequent flows is fundamental in these cases. However, no existing work focuses on finding persistent and infrequent flows. The majority of relevant works aim to either finding frequent flows or persistent ones. In this paper, we are concerned with finding persistent and infrequent flows, $i.e.$, flows that are seen persistently but do not occur that frequently. We call such flows **PI flows**.

B. Prior Art and Limitations

To find PI flows, one straightforward solution is to find the intersection of persistent flows and infrequent flows. The state-of-the-art algorithms for finding persistent flows are the On-Off [2] and the PIE [3], [4]. The state-of-the-art algorithms for estimating flow frequencies are sketch-based algorithms like the Count-Min (CM) sketch [21]. They are typically used to find frequent flows. As the distribution of flow frequencies (also known as flow sizes, the number of packets in a flow) is highly skewed [22], [23] in real network traces, the number of infrequent flows is always very large but considered as less important than frequent ones. To the best of our knowledge, no prior work has focused on finding infrequent flows. Although the above two types of algorithms can find persistent flows and frequent flows respectively, their combination is not optimized for finding persistent and infrequent flows. Indeed, because the set of persistent flows and the set of infrequent flows can be very large, storing both sets leads to large memory consumption, which is unacceptable in network measurement. Large memory consumption also leads to slow speed, because such algorithms often need to run on fast on-chip SRAM (Static RAM) to achieve high speed, and the size of the on-chip SRAM is limited [9], [24]. Here, on-chip memory includes CPU cache, FPGA block RAM [25], $etc.$ In summary, the problem of finding PI flows is new and existing solutions do not work, and we aim to design an efficient algorithm to approach it.

C. Our Solution

In this paper, we propose a novel sketch ($i.e.$, a kind of probabilistic data structures and algorithms), named PISketch, to find persistent and infrequent flows (PI flows) in real time. To the best of our knowledge, this is the first effort to find PI flows. PISketch is compact. For example, it only requires 100KB of memory when working on 10M flows with 4-byte flow ID each. PISketch is accurate. Based on our experiments, the F1 Score of PISketch is around 22.1% and 57.6% higher than two strawman solutions ($i.e.$, On-Off + CM sketch, and PIE + CM sketch), respectively. Also, the Average Relative Error (ARE) of PISketch is on average 820.9 (up to 1188.8) times and 126.2 (up to 265.6) times lower than two strawman solutions, respectively. PISketch is fast. The time complexity for insertion and query is $O(1)$, and its throughput is around 1.23 and 16.5 times higher than two strawman solutions, respectively.

PISketch has two key techniques. The first technique is a Reward and Penalty System that can summarises both persistency and infrequency through one numeric weight; The second technique is a Weight sketch that can find high-weighted flows even if the weight decreases over time:

1) Finding PI flows is much more challenging than finding frequent or persistent flows. The reason behind this is that the traditional weight (frequency/persistency) increases incrementally as time goes by. In contrast, the weight of a PI flow could increase sharply or decrease incrementally. Therefore, the key is to capture the changes of the weight of PI flows. Our first key technique is to design a **Reward and Penalty System** which awards or punishes the weight reasonably. The details are provided in Section III-A.

2) After weighting the PI flows, the challenge lies in how to find the most high-weighted PI flows with limited space. In other words, as new flows arrive continuously, it is a challenge to preserve the old high-weighted flows while taking in new PI flows whose weights have just begun to grow. In this process, the old and new flows will compete fiercely to stay. Since the stayed flows can successfully become potential PI flows, they can get more opportunities to be observed. There is no existing work can directly handle the top-k weight problem whose weight could decrease. Thus, we propose our second key technique in the Weight sketch, called **Weight Fusion Strategy**, which decrements the low weight flows to make room for new flows (see Section III-B - III-C for details).

We provide strict theoretical derivations, see Section IV for details. We implement PISketch entirely on P4 and FPGA platforms in Section V. Further, we conduct extensive experiments on CPU platform, and our experimental results demonstrate the obvious advantages of PISketch over two strawman solutions. In addition, we implement two concrete cases of PISketch for the preliminary detection of APT and FRP flows. More details are provided in Section VI. We provide all the related code open-source at GitHub [26].

Key Contributions:

- We propose and define a new problem called "finding persistent and infrequent flows", which has not been studied before.
- We propose a novel sketch, PISketch, to find persistent and infrequent flows, accurately, fast, and using limited memory.
- We mathematically analyze PISketch to prove its accuracy and space-time efficiency by theoretically deriving its error bounds and memory and time cost.
- We fully implement PISketch on P4 and FPGA, and conduct extensive experiments on CPU. Experimental results show that our PISketch outperforms two strawman solutions (On-Off + CM sketch, and PIE + CM sketch). In addition, we also implement two concrete cases of PISketch through end-to-end experiments.

II. Related Work

A. Finding Persistent Flows

Several algorithms have been proposed to find persistent flows [2], [3], [4], [5], [6], [27], [28]. The state-of-art

algorithms are On-Off [2] and PIE [3], [4]. The idea of On-Off is to exploit the increasing persistence of flows. No matter how many flows are mapped to the same counter in a time window, On-Off only increases this counter by one. In this way, On-Off first estimates the persistence of all flows, and then changes the data structure to split persistent and non-persistent flows. It only stores the information about persistent flows, and protects them from replacements and hash collisions with other flows. PIE uses a data structure called Space-Time Bloom filter based on the Invertible Bloom filter [29], [30] and a Raptor code [31] to encode the flow IDs. During each measurement period, PIE maintains a Space-Time Bloom filter. When inserting a flow, it uses the Raptor code to encode the flow ID into many segments, and randomly selects some segments to store in the Space-Time Bloom filter, which aims at reducing the memory usage. When querying a flow, PIE gathers all Space-Time Bloom filters. If and only if it occurs in enough measurement periods and enough encoded bits for the stored ID, PIE can decode its flow ID from the Space-Time Bloom filter.

B. Frequency Estimation

Frequency (flow size) estimation consists of estimating the number of occurrences of flows. It has been widely studied. Sketches have proved their superiority in frequency estimation [11], [12], [32], [33]. Actually, they can achieve high accuracy and speed with limited memory usage. There are many sketch-based algorithms for estimating flow frequency [7], [9], [10], [21], [34], [35], [36], [37], [38], [39], [40], the most typical being the widely used Count-Min (CM) sketch [21]. The CM sketch uses multiple equal-sized buckets. Every bucket is associated with a hash function h_i. When a flow f arrives, every bucket calculates its hash value $h_i(f)$ to map f to the cell $A[i][h_i(f)]$, and the value of the cell is increased by 1. For the recorded flow, its frequency is the minimum value of all its mapped cells, and then top-k frequent flows can be selected. Besides the CM sketch, other typical sketch-based algorithms include sketches of CU [34], C [35], CSM [36], and ASketch [37], PyramidSketch [38], HeavyKeeper [10], HeavyGuardian [39], Cold Filter [40], *etc.* which focus on accurately estimating large/elephant flows. State-of-the-art sketch-based network measurement systems include SketchVisor [8], UnivMon [7], ElasticSketch [9], Nitrosketch [11], CocoSketch [12], LightGuardian [41], *etc.*

C. Membership Query

The most typical algorithm for membership query is Bloom filter [42]. A standard Bloom filter is a highly compact probabilistic structure that consists of an $\mathcal{M}$ bits array with k hash mapping functions: $h_1(.), h_2(.), \ldots, h_k(.)$, and each bit is set to 0 at the beginning. For each incoming flow, its k mapped bits are set to 1. For a membership query, *i.e.*, querying whether a flow occurs in the data stream, the Bloom filter checks whether all its k mapped bits have been set to 1. The above makes Bloom filter very good at "removing duplicates": if the k mapped bits are all 1, it means the flow has already appeared/repeated. Some successors include Counting Bloom Filter [43], Spectral Bloom Filters [44], Dynamic Bloom Filter [45], Variable-Increment Counting Bloom Filter [46], and Elastic Bloom Filter [47], *etc.*

III. PISketch Design

In this section, we first define the weight of a flow in Section III-A. Then, we introduce the data structure of PISketch in Section III-B. Next, we give the details of how to process incoming flows based on the weight in Section III-C. For convenience, we provide the symbols frequently used in this paper and their meanings in Table I.

A. Weight Definition

Preliminary: Given a data stream $\mathcal{S}$, we divide it into V equal-sized and continuous time windows.

We use the same weight definition for each time window. If flow f appears in the i^{th} window for the first time, its weight W_i is incremented by the initial value L; when f appears in this window for the second time, its weight is decremented by 1; when f appears in this window for the third time, its weight is also decremented by 1, and so on.

Initially, the weight $W_i\,(1 \le i \le V, i \in \mathbb{Z}^+)$ of flow f that occurs $O_i\,(O_i \in \mathbb{Z}^+)$ times in the i^{th} window can be calculated as:

$$W_i = \begin{cases} 0, & \text{if } O_i = 0; \\ L - (O_i - 1), & \text{if } O_i \ge 1. \end{cases} \tag{1}$$

where O_i is the number of occurrences of flow f in the i^{th} window. If $O_i = 0$, flow f never occurs in the window, so we set $W_i = 0$. $L\,(L \in \mathbb{Z}^+)$ is the initial value assigned to each flow when it first appears in the i^{th} window.

Equation 1 is the mathematical expression of our proposed *Reward and Penalty System*. Next, we define the total weight W_f of flow f in all windows as $W_f = \sum_{i=1}^{V} W_i$.

PISketch design goal: Persistent and infrequent flows (PI flows) refer to the flows whose total weight is larger than a given threshold $\mathcal{T}$. Actually, $\mathcal{T}$ can be defined by the users according to their requirements or the specific application requirements. Flows with higher total weight are more likely to be reported as PI flows. Note that the total weight of a flow may be negative, in which case the flow is definitely not among the PI flows we want to report, due to its high occurrences.

B. Data Structure

As illustrated in Figure 1, PISketch consists of two parts: The first part reports whether a flow occurs in current time window for the first time. The second part calculates the weight of each flow and finds out which flows are most likely to have high weight.

The data structure of the first part is a Bloom filter [42] (see Section II-C). The Bloom filter is a compact representation of the flows that have arrived in the current time window. When a new time window begins, we reset the Bloom filter. Every time a flow comes, we query whether it has been seen for the first time (*i.e.*, has not been inserted), and if so then insert it into the Bloom filter. Then, we pass the answer to

TABLE I
SYMBOLS FREQUENTLY USED IN THIS PAPER

Symbol	Meaning
$\mathcal{S}$	A data stream with many flows
f	An arbitrary flow in $\mathcal{S}$
V	Number of divided windows
B_i	i^{th} bucket
U	Number of buckets
p	Number of cells in a bucket
W_i	Weight of flow f in the i^{th} window
W_f	Total weight of f
f'	The flow with the smallest weight in the mapped bucket
$W_{f'}$	The smallest weight in the mapped bucket
O_i	Number of occurrences of f in the i^{th} window
L	Initial value given when f first enters any window
$\mathcal{T}$	A user-defined threshold
N_f	Total number of windows that f occurs
N_{Total}	Total number of flows
N_{PI}	Number of PI flows

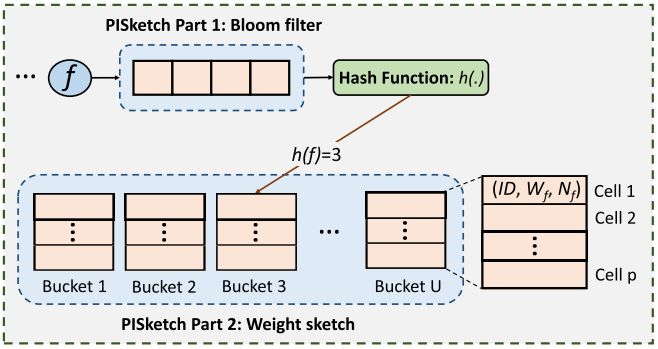


Fig. 1. Data structure of PISketch.

the next part for weight calculation. A Bloom filter is used to remove duplicates from incoming flows. Removing duplicates is necessary because the operations for the first arrival and subsequent arrivals are different, according to Equation (1). If the Bloom filter reports true, it means that the flow has appeared in this time window.

The data structure of the second part is the *Weight sketch*. Its basic structure is a hash table with U buckets $B_1, B_2, \ldots, B_U$. Based on the query results in the previous part, the Weight sketch calculates the total weight of each flow according to Equation (1) and keeps the flows whose weights might potentially exceed $\mathcal{T}$ as much as possible. Each bucket of the Weight sketch has p cells, and each cell has three fields including flow ID (key), (total) weight and the number of windows where the flow occurs. A hash function $h(.)$ also randomly maps the flow to one of the buckets.

C. Operations

Insertion: Given an incoming flow f to the i^{th} window, we first query the Bloom filter to check whether this flow has occurred in the current window. If the Bloom filter reports false, indicating f does not occur in the current window, then we insert f into the Bloom filter and perform two operations for this window: initialize the weight $W_i = L$ and increment window number $N_f = N_f + 1$. Otherwise, it indicates that flow f has already occurred in the current window. We decrement the weight W_i of this window by 1, *i.e.*, $W_i = W_i - 1$, as shown in Equation (1). Then, we try to store the information of f to its mapped bucket $B_{h(f)}$. According to the content of $B_{h(f)}$, there are three different cases:

Case 1: If a cell contains f, we update the fields of this cell: (1) We add W_i to the total weight W_f, *i.e.*, $W_f \leftarrow W_f + W_i$; (2) We update the stored N_f to the current one.

Case 2: If we fail to find f in $B_{h(f)}$ and $B_{h(f)}$ is not full, then we store f in an arbitrary empty cell. We set W_f to W_i, *i.e.*, $W_f \leftarrow W_i$. In this case: $W_f = L$, $N_f = 1$.

Case 3: If no cell in $B_{h(f)}$ contains f and $B_{h(f)}$ does not contain empty cells, then we try to evict a flow from $B_{h(f)}$ to make room for f. To keep as many potential PI flows as possible in the data structure, we select a flow f' whose weight is the smallest among all flows of $B_{h(f)}$. Although the weight of f' is the smallest, we cannot determine yet whether f' is a PI flow or not. Therefore, we evict f' with the smallest weight $W_{f'}$ in $B_{h(f)}$ using the *Weight Fusion Strategy*: Whenever an eviction/replacement happens, we decrement $W_{f'}$ by 1. After decrementing, if $W_{f'}$ is lower than 0, it means that the replacement is successful. We then evict flow f', and f occupies the position of f'. W_f is set to $W_i + 1$, and N_f is set to 1. If the replacement is unsuccessful, f leaves.

Finally, we clear the Bloom filter by setting all bits to 0 at the end of each time window. The pseudo-code of the insertion operation is shown in Algorithm 1.

Query: Based on the above operations, PISketch can keep many PI flows with high weights. To get these PI flows, PISketch only needs to traverse the buckets. Note that all the reported flows are potential PI flows. Therefore, users should carry on further analysis of these potential PI flows. Furthermore, the number of reported PI flows depends on the memory size of the data structure. The minimum memory size of the data structure should therefore be adapted to the minimum expected number of PI flows.

Examples: As shown in Figure 2, we set $L = 5$, $U = 3$, and $p = 2$. We assume that flows e, r, g are coming to this

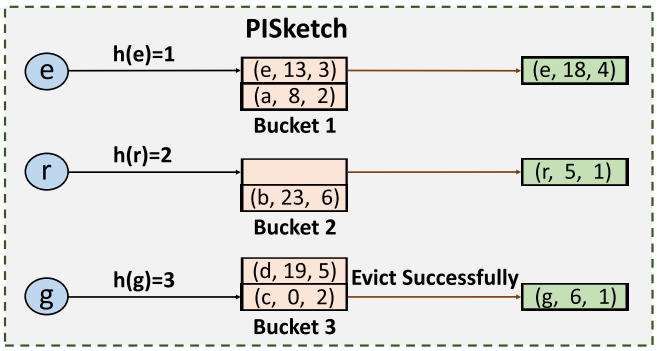


Fig. 2. Examples of processing flows.

Algorithm 1 Insertion Procedure

Input: a incoming flow f
search for f in the Bloom filter;
if f *is in the Bloom filter* **then**
  $W_i = -1$;
else
  insert f into the Bloom filter;
  $W_i = L$;
search for f in bucket $B_{h(f)}$;
if f *is in* $B_{h(f)}$ **then**
  $W_f \leftarrow W_f + W_i$;
  if $W_f < 0$ **then**
    evict f;
else
  if *the bucket has empty cells* **then**
    allocate a cell for f;
    $W_f \leftarrow W_i$;
  else
    // let f' be the minimum weight flow in $B_{h(f)}$;
    $W_{f'} \leftarrow W_{f'} - 1$;
    if $W_{f'} < 0$ **then**
      replace f' with f;
      $W_f \leftarrow W_i + 1$;
if f *is the last flow in the current window* **then**
  empty the Bloom filter;

window for the first time. When receiving a flow e, PISketch maps it to B_1 through the hash function $h(e)$. Because there is a cell storing e, PISketch directly updates the three fields of the cell: PISketch increments the W_e by 5, and increments the N_e by 1. When receiving a flow r, PISketch maps it to B_2. Note that r is not stored in B_2, and there is an empty cell. PISketch stores r in the empty cell: set the ID, W_r, N_r to the ID of r, 5 and 1, respectively. When processing flow g, PISketch first calculates the hash function $h(g)$ to locate bucket B_3. Because none of the cells in B_3 stores g, PISketch tries to evict a flow stored in B_3 to make room for g. Two flows are stored in B_3, and the weight of d is obviously larger than the one of c. Therefore, PISketch decrements W_c by 1, and W_c becomes -1. g now occupies the cell storing c. PISketch stores g in this cell: set the ID, W_g, N_g to the ID of g, 6 and 1, respectively.

IV. Mathematical Analysis

In this section, we first analyze the property of the weight of a flow in Section IV-A. Then, we derive error bounds for the Bloom filter and Weight sketch in Section IV-B and Section IV-C, which are the data structures of Part 1 and Part 2 of PISketch, respectively. Finally, we analyze the time complexity of PISketch in Section IV-D.

Based on Equation (1), if a flow occurs O_i times in a single window, then its weight in this window W_i can be calculated as:

$$W_i = \begin{cases} 0, & \text{if } O_i = 0; \\ L + 1 - O_i, & \text{if } O_i \geq 1. \end{cases} \tag{2}$$

TABLE II
Symbols Used in Section IV

Symbol	Meaning
V	Number of divided windows
$\mathcal{T}$	A user-defined threshold
f	An arbitrary flow in $\mathcal{S}$
λ	Poisson parameter of f
q	binomial parameter of f
W_f	Total weight of f reported by PISketch
$\hat{W}_f$	Real weight of f
Γ	Approximate number of flows in each window
m	Memory cost for Bloom filter
U	Number of buckets

In this section, we analyze the implementation of PISketch and provide a theoretical bound. We make the following two assumptions:

1) The majority of the flows in the data stream are neither persistent nor frequent flows (see Table V in Section VI-A for related evidence); another proportion of flows are both persistent and frequent flows, and PI flows only make up a small quantity of data streams.
2) For all PI flows, there is a probability of $1 - q$ that it does not appear in this window, and a probability of q that its appearance in this window, following a Poisson distribution with parameter λ. In other words, let $X \sim B(1, q)$ and $Y \sim P(\lambda)$ be two independent random variables, then O_i can be written as

$$O_i = (Y + 1) \cdot 1_{\{X=1\}}. \tag{3}$$

Here, we use $Y + 1$ instead of Y to guarantee at least one occurrence in a given window.

The symbols used in this section is listed in Table II.

A. Property of the Weight of a Flow

In this part, we first give the expectation of the weight when the frequency of the flow satisfies a special distribution. Then, we claim the limited extent of over-estimation error, which shows that the precision rate (PR) of PISketch is usually close to 1.

We first show the expectation of the weight when the frequency in each window follows a special distribution in an ideal situation.

Theorem 1: If the weight changes due to some unsuccessful replacement or false positives in the Bloom filter are ignored, then the expectation of the weight after a window is given by the following equation:

$$\mathbb{E}W = q(L - \lambda). \tag{4}$$

Proof: Let O denote the number of occurrences in a window. Since

$$\mathbb{P}(O = k+1) = q\frac{\lambda^k}{k!}e^{-\lambda}, (k \geq 0) \tag{5}$$

the expectation of its weight can be written as

$$\begin{aligned}\mathbb{E}W &= q\sum_{k=0}^{+\infty}(L-k)\frac{\lambda^k}{k!}e^{-\lambda} \\ &= qL\sum_{k=0}^{+\infty}\frac{\lambda^k}{k!}e^{-\lambda} - q\sum_{k=0}^{+\infty}k\frac{\lambda^k}{k!}e^{-\lambda} \\ &= qL\sum_{k=0}^{+\infty}\frac{\lambda^k}{k!}e^{-\lambda} - q\lambda\sum_{k=1}^{+\infty}\frac{\lambda^{k-1}}{(k-1)!}e^{-\lambda} \\ &= q(L-\lambda). \end{aligned} \tag{6}$$

□

Based on the theorem above, a reasonable explanation for the parameter q and λ is that q measures the **persistence** of a flow: for a large q, the probability of a flow appearing in each window will be high as well, which illustrates its persistence; λ measures the **frequency** of a flow: for a larger λ, the flow tends to appear more times in a single window, which relates to its frequency. As a result, to guarantee a sufficiently high expectation of the weight in a single window, a PI flow shall have a large q and a small λ with respect to its occurrence.

Moreover, if $\lambda > L$ for a flow f, by Kolmogorov's Strong Law of Large Numbers, we know that

$$\mathbb{P}\left(\lim_{V\to\infty}\frac{W_f}{V} = q(L-\lambda)\right) = 1. \tag{7}$$

Hence the total weight of f converges almost surely to a negative value. We conclude that these flows are definitely **not** PI flows we want to find.

Theorem 2 (Claim of Limited Over-Estimation Error): If f has not been evicted from the Weight sketch, then $W_f \leq \hat{W}_f + 1$.

Proof: The weight of flow f will be incremented by at most 1 when f successfully replace a flow in the Weight sketch, but may be decremented by 1 every time a flow not in the Weight sketch comes to the bucket. Also, the false positives brought by the Bloom filter will wrongly change W_i from L to -1, which potentially decrements W_f as well. Hence, $W_f \leq \hat{W}_f + 1$. □

The theorem above points out that PISketch tends to under-estimate the weight of a flow. Only under the following circumstance will $W_f > \hat{W}_f + 1$: leftmargin=1em

- f appears a sufficiently large number of times in a given window. As a result, it will be evicted from the Weight sketch first, which makes every decrement operation on f afterwards in vain (f is not in the bucket, so there is nowhere to decrement its weight).
- f finally enters the Weight sketch, so its weight will be reported by PISketch.

Since f appears many times in a single window, f is definitely not a PI flow we want to find. As a result, we conclude that for every PI flow f, $W_f \leq \hat{W}_f + 1$. In other words, the over-estimation error of PISketch is limited, and the precision rate of PISketch is always close to 1.

B. Error Bound of the Bloom Filter

In this part, we work out the error caused by the Bloom filter [42]. We use $\tilde{W}_f$ to denote the total weight of f reported by PISketch with a perfect Bloom filter.

Theorem 3: Let δ be the false positive rate (FPR) of the Bloom filter, then

$$0 \leq \mathbb{E}\tilde{W}_f - \mathbb{E}W_f \leq \delta q V(L+1). \tag{8}$$

Proof: The expectation of the number of windows that f will appear is qV, and every false positive will change W_i to -1 instead of L. Since the false positive rate is δ,

$$0 \leq \mathbb{E}\tilde{W}_f - \mathbb{E}W_f \leq \delta q V(L+1). \tag{9}$$

□

It is already known that $m = O\left(\Gamma \log \frac{1}{\delta}\right)$. Hence, for a given constant $C > 0$, by Markov's inequality we get

$$\mathbb{P}\left(\tilde{W}_f - W_f \geq C\right) \leq \frac{\mathbb{E}(\tilde{W}_f - W_f)}{C} = \frac{qV(L+1)}{C}e^{-O\left(\frac{m}{\Gamma}\right)}. \tag{10}$$

By allocating more space for the Bloom filter, the probability above can be bounded.

C. Error Bound of the Weight Sketch

In this part, we focus on the error brought by the Weight sketch. We first define the idea of safe flows. Then, we analyze the memory cost to ensure that a PI flow will become "safe" and give a property for PI flows which has its weight less than the threshold.

Before we give an error bound for the Weight sketch, we take a careful look at a PI flow f after it enters the Weight sketch: f is expected to enter the Weight sketch with its initial weight $W_f = L$ (if there is an empty cell in the bucket), $W_f = L+1$ (if f replaces some flow in the bucket and f appears for the first time in this window) or $W_f = 0$ (if f replaces some flow in the bucket and f does not appear for the first time in this window). Since most flows in the data stream are non-persistent and infrequent, we assume that they appear in a single window in the data stream and the weight of most non-persistent and infrequent flows in the Weight sketch is at most $L+1$. Hence, f is most likely to be cleared from the Weight sketch when $W_f \leq L+1$. Once $N_f \geq 2$ and $W_f > L+1$, f will stand out in the bucket and will hardly be evicted from the Weight sketch. Since flows with higher weight are unlikely to be evicted due to the replacement strategy, we define safe flows as follows:

*Definition 4: A PI flow f, is called **safe** in the time window i if $W_f > L+1$ at the end of time window i.*

Theorem 5: Assume that the number of cells in each bucket is kept constant. If a PI flow f has entered the Weight sketch, to ensure that the probability of f being safe is not less than $1-\varepsilon$, the Weight sketch needs $U = O\left(\Gamma \log \frac{1}{\varepsilon}\right)$ buckets.

Proof: The main idea for the proof is that more space for the Weight sketch will bring more low-weight flows to the bucket. On this occasion, the replacement strategy will influence these victims and allow for the infrequency of flow f. We will just give a proof based on an average case for

simplicity. Assume that all flows are distributed into each bucket evenly. As a result, approximately $U(L+1)$ flows will lead to $L+1$ flows coming to a single bucket, which can evict one flow in the bucket. If our Weight sketch unluckily chooses f for eviction, then f will be driven out of the bucket. These $U(L+1)$ flows consist of approximately $\frac{U(L+1)}{\Gamma}$ windows, and an evicted f shall not appear in these windows. Hence, we get

$$(1-q)^{\frac{U(L+1)}{\Gamma}} \leq \varepsilon \Rightarrow U \geq \frac{\Gamma}{L+1} \cdot \frac{\log \frac{1}{\varepsilon}}{\log \frac{1}{1-q}}. \quad (11)$$

Here, q is determined by f, and L is a parameter usually defined by users, so we get $U = O\left(\Gamma \log \frac{1}{\varepsilon}\right)$. □

The theorem above gives a conservative estimation of the probability, as the Weight sketch will choose the flow with the smallest weight for replacement and the existence of some frequent flows will make PI flows more difficult to be evicted. Hence, a PI flow f which does not appear in many windows in a row can still be kept in the Weight sketch with a high weight.

As to safe flows, there is a high probability that the total weight of f exceeds the threshold $\mathcal{T}$.

Theorem 6: For a PI flow f, assume that its weight is never changed due to unsuccessful replacement and false positives never occurs with respect to the Bloom filter. If the Bloom filter reports $W_i = L$ for n times before it successfully enters the Weight sketch and becomes safe and finally $W_f < \mathcal{T}$, then $n > \frac{\hat{W}_f - \mathcal{T}}{L}$.

Proof: f shall enter the Weight sketch very late to get a relatively small weight. Every ignored appearance reported by the Bloom filter will subtract L from its weight. Hence $\tilde{W}_f = \hat{W}_f - nL$. With a perfectly accurate Bloom filter, $\tilde{W}_f = W_f$, hence

$$W_f = \tilde{W}_f = \hat{W}_f - nL < \mathcal{T} \Rightarrow n > \frac{\hat{W}_f - \mathcal{T}}{L}. \quad (12)$$

□

Although the Bloom filter cannot be perfectly correct in practice, Equation (10) offers a solution to control the error caused by it, hence the theorem above still applies to the data stream in practice.

D. Time Cost of PISketch

Theorem 7: For each flow in the data stream, the time cost to handle it is $O(1)$.

Proof: For each flow in the data stream, PISketch first checks the Bloom filter, then uses a hash function $h(.)$ to map the flow into one of the buckets. In insertion operation, PISketch either updates it in the bucket or tries to replace a flow in the bucket. Since the number of cells in each bucket is small, all these operations can be done in constant time. At the end of every window, PISketch empties the Bloom filter, the amortized cost of which is $O(1)$ for each flow. Hence, the total time cost for each flow is $O(1)$. □

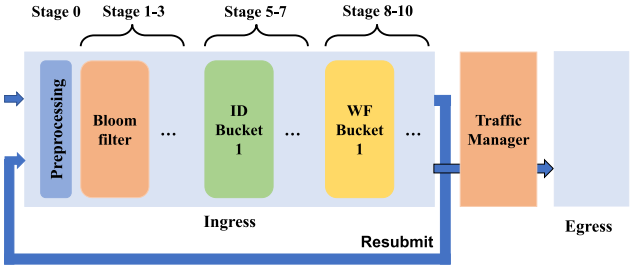


Fig. 3. Architecture design of the Tofino version of PISketch.

V. Implementation

In this section, we implement and evaluate PISketch on the Tofino switch and FPGA, presented in Section V-A and Section V-B, respectively.

A. Evaluation on Tofino Platform

We have fully built a P4 prototype of the PISketch on the Tofino switch [48] using P4 language [49]. In the Tofino version of PISketch, only the ID and weight of the flow are reserved for each cell in the Weight sketch (Part 2) to ensure sufficient hardware resources.

Challenge: In Tofino architecture, packets go through the ingress pipeline, the traffic manager and the egress pipeline in turn. The Ingress and egress pipeline each contains 12 separate stages, each capable of handling some simple logic operations and having independent memory. Due to the speed requirements of the switch, packets can only visit each stage once, so finding the minimum value in the bucket and modifying it becomes the biggest challenge in our implementation. To address the above challenge, we use Tofino built-in resubmit function, which means that a packet will be resent to the ingress port after passing through the ingress pipeline, and then go through the ingress pipeline again, accessing the memory of each stage for a second time, so as to find the minimum value and modify the buckets.

Design: As shown in Figure 3, we use a total of 11 stages in Ingress to implement the P4 version of PISketch under the Tofino model. Specifically, Stage 1 to 3 are Bloom filters, Stage 5 to 7 are 3 buckets that store flow IDs, and Stage 8 to 10 are 3 buckets that store the weights W_f corresponding to flow IDs. Each stage holds a bucket array of length 2^{17}, and each bucket holds 32-bit data in width. As packets pass through, the data corresponding to the index of the array is accessed and logically manipulated using the hash result of the flow ID as the index. Each packet passes through the ingress pipeline twice. The key operation of the first round is to complete the reading of each stage data. When packets enter the switch, they go through Bloom Filter and record the query result in packet header. Then the packets read three group of ID buckets and WF buckets in turn to find whether there is an ID match and record the bucket number with the smallest WF value. After finishing the process, the switch resubmits the packets to the ingress port. The key operation of the second round is the processing of writes to empty buckets or buckets whose flow IDs are matched or buckets with the smallest weight W_f among them. Packets resubmitted to the

TABLE III
H/W RESOURCES USED BY PISKETCH

Resource	Usage	Percentage
Hash Bits	597	11.96%
SRAM	184	19.17%
Map RAM	184	31.94%
TCAM	0	0%
Stateful ALU	14	29.17%
VLIW Instr	27	7.36%
Match Xbar	113	7.10%

ingress port will go through the ingress pipeline again. Based on the query results of the first round, PISketch performs the operations of ID replacement or WF increase or decrease. When finishing the two round, the packets enter the traffic manager, pass the egress pipeline and leave from the switch.

In practice, we need two sets of exactly the same PISketch structure. The old and the new PISketch are switched while the time window is switched. When we use the new PISketch, the control plane records the old PISketch results and clear the old data.

Evaluation Results: We list the utilization of various hardware resources on the switch in Table III. We find that Map RAM and Stateful ALU are the two most used resources of PISketch, accounting for 31.94% and 29.17% of the total quota, respectively. For other kinds of sources, PISketch uses up to 19.17% of the total quota.

The Tofino switch ensures the strict order of entry into the ingress pipeline, which means the order of packets passing through each stage is the same as the entering sequence. The CPU environment can easily simulate the behavior of the Tofino Ingress, so the Tofino test results can be illustrated by the CPU simulation experiments. We do not show them here repeatedly.

B. Evaluation on FPGA Platform

We implement PISketch on an FPGA network experimental platform (Virtex-7 VC709). The FPGA integrated with the platform is xc7vx690tffg1761-2 with 433200 Slice LUTs, 866400 Slice Register, and 1470 Block RAM Tile.

Design: The architecture design diagram is shown in Figure 4. The FPGA version of PISketch follows a hierarchical and modular design concept. The top-level module implements the overall input and output of the system, and controls the parameter transfer between the following three sub-modules: calculating the hash values (Hash), checking the number of times the flow arrives at the window (Bloom filter), and writing to the matched bucket (Weight sketch). Among them, since the Bloom filter module needs to be cleared periodically, we choose to use the register to store the Bloom filter. However, larger registers result in lower clock frequency, which is the performance bottleneck of FPGA-based PISketch. Also, the clearing of RAM requires RAM depth sub-clock cycles, so we have to sacrifice performance to ensure complete functionality. In addition, the matching bucket in the Weight

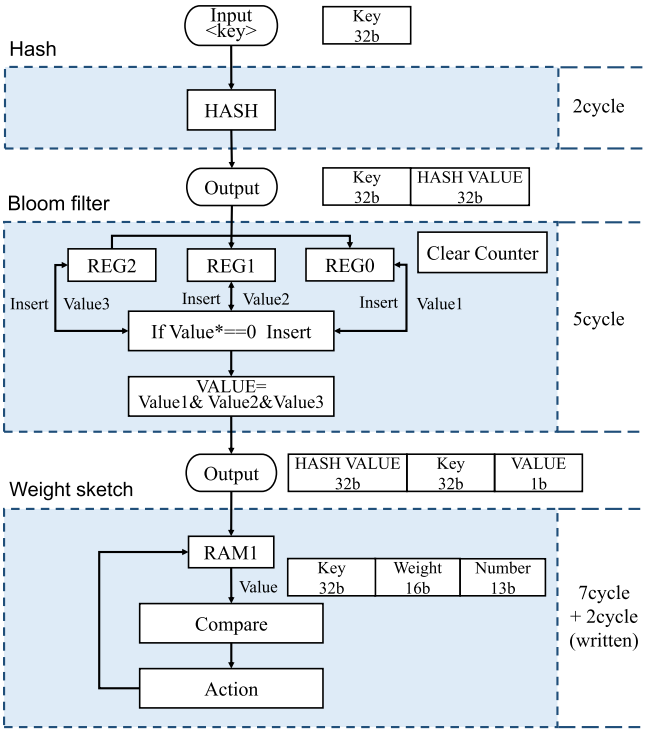


Fig. 4. Architecture design of the FPGA version of PISketch.

sketch module chooses RAM for storage and involves the operation of reading the relevant data from RAM and then comparing it with the current flow. However, the RAM used in this experiment requires two clock cycles for read and write operations, so the module uses FIFO (First Input First Output) to process the current flow after two clock cycles in order to achieve the goal of full pipeline. In short, our FPGA-based PISketch makes full use of hardware parallel processing to achieve full pipeline operation. The processing of each flow requires 16 clock cycles, and each clock cycle completes the corresponding 1/16 subtask, thus 16 flows can be processed in parallel per clock cycle.

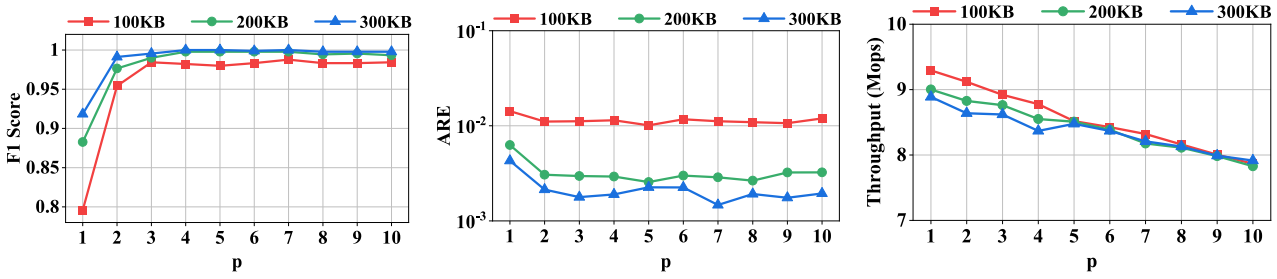
Evaluation Results: The evaluation results of the FPGA implementation are shown in Table IV. 1) The overall resource usage information is as follows: ① PISketch uses 11745 LUTs, 2.71% of the 433200 total available; ② PISketch uses 3985 Register, 0.46% of the 866400 total available; ③ PISketch uses 900 Block RAM Tile, 61.22% of the total on-chip Block RAM. Specifically, the resource usage and percentages of sub-modules (Hash, Bloom filter, and Weight sketch) correspond to the numbers in the table and the ones in parentheses beside them, respectively. 2) The clock frequency of our implementation in FPGA is 276 MHz, meaning the processing speed (*i.e.*, throughput in Section VI-B) of the system can be 276 Mips (million items per second).

VI. EXPERIMENTAL RESULTS

In this section, we show the experimental results of PISketch on CPU. First, we describe the experimental setup and metrics in Section VI-A and Section VI-B, respectively. Second,

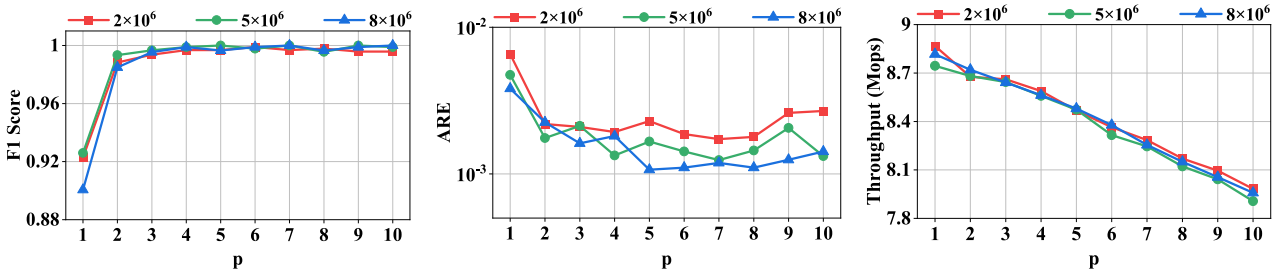
TABLE IV
PERFORMANCE ON FPGA PLATFORM

Module	Resource Overhead			Frequency (MHz)
	LUTs	Register	Block RAM Tile	
Hash	94 (0.02%)	130 (0.02%)	0 (0.00%)	276
Bloom filter	7440 (1.72%)	3350 (0.39%)	2 (0.14%)	276
Weight sketch	4211 (0.97%)	505 (0.06%)	898 (61.09%)	276
Total	11745 (2.71%)	3985 (0.46%)	900 (61.22%)	276



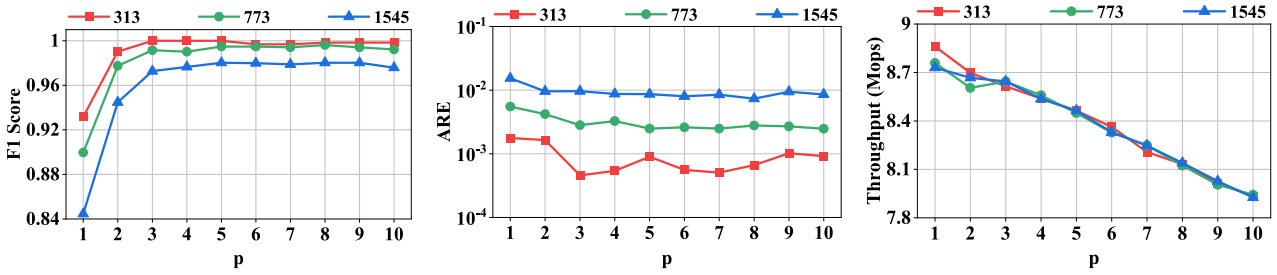
(a) F1 Score (b) ARE (c) Throughput

Fig. 5. Evaluation on parameter settings: p, with different memory sizes.



(a) F1 Score (b) ARE (c) Throughput

Fig. 6. Evaluation on parameter settings: p, with different N_{Total}.



(a) F1 Score (b) ARE (c) Throughput

Fig. 7. Evaluation on parameter settings: p, with different N_{PI}.

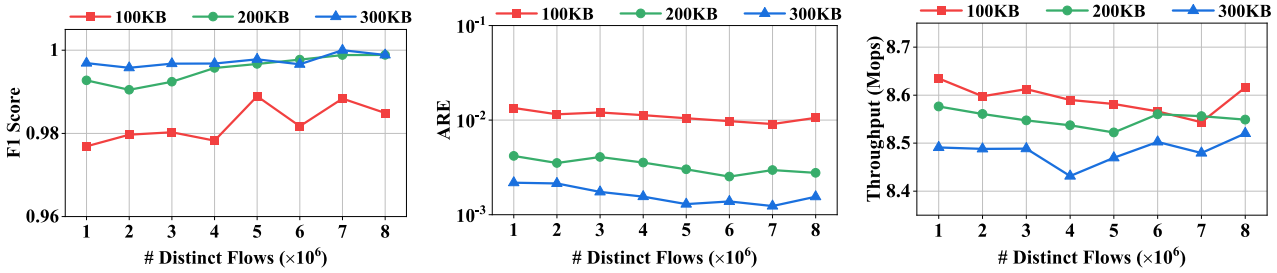
we explain how parameter settings affect PISketch's performance in Section VI-C. Third, we evaluate the performance of PISketch on different datasets and compare it with two strawman solutions in Section VI-D. Then, we provide two concrete applications of PISketch through two end-to-end experiments in Section VI-E and Section VI-F, along with a discussion about them in Section VI-G. Finally, we provide examples of parameter settings for problem definition and an open question in Section VI-H and Section VI-I, respectively.

A. Experiment Setup

Implementation: We implement our algorithm and related algorithms in C++. Our hash function is the Bob Hash [50]. **We conduct experiments on a server with two CPUs (Intel Xeon E5-2620V3@2.4GHZ) and 62GB DRAM.**

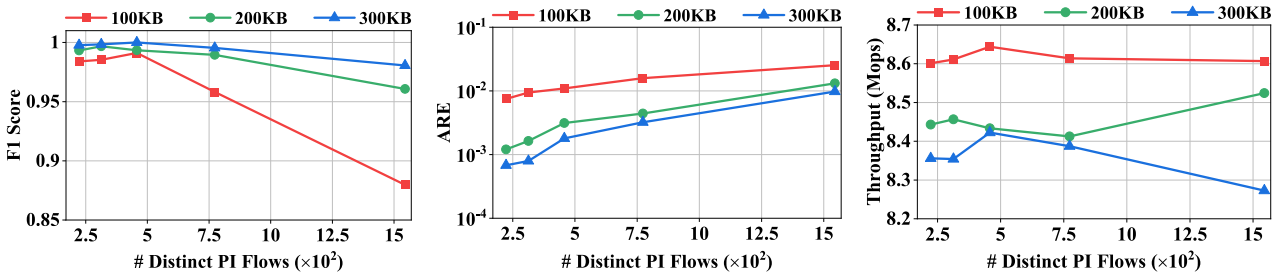
Datasets: We use three real-world datasets and two synthetic datasets. Each dataset contains about 5M packets.

(1) CAIDA Dataset: This IP Trace Dataset is streams of anonymized IP traces collected in 2018 by CAIDA [51].



(a) F1 Score (b) ARE (c) Throughput

Fig. 8. Evaluation on parameter settings: N_{Total}, with different memory sizes.



(a) F1 Score (b) ARE (c) Throughput

Fig. 9. Evaluation on parameter settings: N_{PI}, with different memory sizes.

(2) MAWI Dataset: This real packet traffic trace dataset is provided by the MAWI Working Group [52].

(3) Network Dataset: This dataset contains users' posting history on the stack exchange website [53]. Each flow has three values u, v, t, which means user u answered user v's question at time t. We use u as the ID and t as the timestamp of a flow.

Here, we list the number and percentage of persistent flows as well as frequent flows in the three real-world datasets mentioned above in Table V. We can further draw the following conclusions: the proportion of persistent flows is very small, and the proportion of PI flows, which must also be infrequency flows under this premise, is even less.

(4) Synthetic Dataset I: Since large-scale real APT flows are too difficult to obtain, we synthesize this dataset by referring to some literature [54], [55], [56], [57] to evaluate the performance of PISketch's preliminary screening for suspected APT flows. Specifically, our synthetic approach consists of mixing the real APT flows from Contagio Malware Database [58] with the normal flows from CAIDA Dataset. In this dataset, about 600 real attack flows are mixed with millions of normal flows.

(5) Synthetic Dataset II: Since large-scale real FRP flows are too difficult to obtain, we synthesize this dataset to evaluate the performance of PISketch's preliminary screening for suspected FRP flows. Our synthetic approach is similar to the above Synthetic Dataset I, except that the real FRP flows are collected by ourselves (see the **Methodology** in Section VI-F). In this dataset, about 40 captured FRP flows are mixed with millions of normal flows.

B. Metrics

We evaluate the following three performance metrics: F1 Score, Average Relative Error (ARE) and Throughput.

(1) F1 Score: $\frac{2*PR*RR}{PR+RR}$. Precision Rate (PR) indicates the ratio of truly reported PI flows to the reported flows, and Recall Rate (RR) indicates the ratio of truly reported PI flows to the total PI flows. We use F1 Score to evaluate the accuracy.

(2) Average Relative Error (ARE): Let $\hat{N}_1, \hat{N}_2, \ldots, \hat{N}_k$ be the estimated window number of the reported flows, and let $N_1, N_2, \ldots, N_k$ be the true window number of the reported flows. ARE is defined as $\frac{1}{k} \cdot \sum_{j=1}^{k} \frac{|N_j - \hat{N}_j|}{N_j}$.

(3) Throughput: Million operations (insertions) per second (Mops). All the experiments about throughput are repeated 10 times, and the average throughput is reported. We use throughput to evaluate the speed.

C. Experiments on Parameter Settings

We have four key parameters, which can be divided into the following two types:

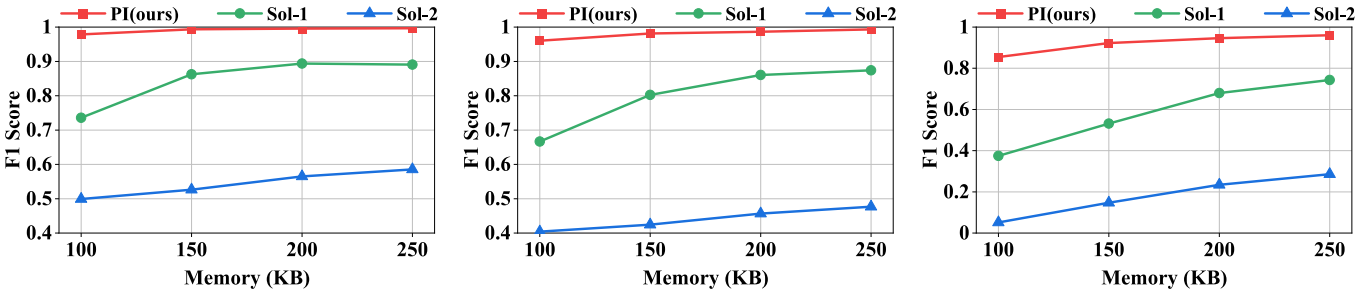
(1) Parameters related to the problem definition: They include the initial value L, the number of windows V, and the threshold $\mathcal{T}$. These parameters are usually defined by users according to different use cases. In this experiment, we set $L = 10$, $V = 1000$ and $\mathcal{T} = 3000$. We recommend this set of parameters because of their good filtering performance across different datasets. Please refer to Section VI-H for details.

(2) Parameters related to the algorithm setting: They include the number of cells in a bucket p, the total flow number N_{Total}, and the number of PI flows N_{PI}. We use the CAIDA dataset, and use F1 Score, ARE and Throughput as metrics to evaluate its effects.

Effects of p (Figure 5-7): We find that the PISketch achieves the best F1 Score when the number of cells in a bucket is 4 and 5. 1) In this experiment, we first compare the performance of the PISketch when p varies from 1 to 10 and with different memory sizes, as shown in Figure 5(a)-5(c). When the memory size is 200/300KB, the F1 Score

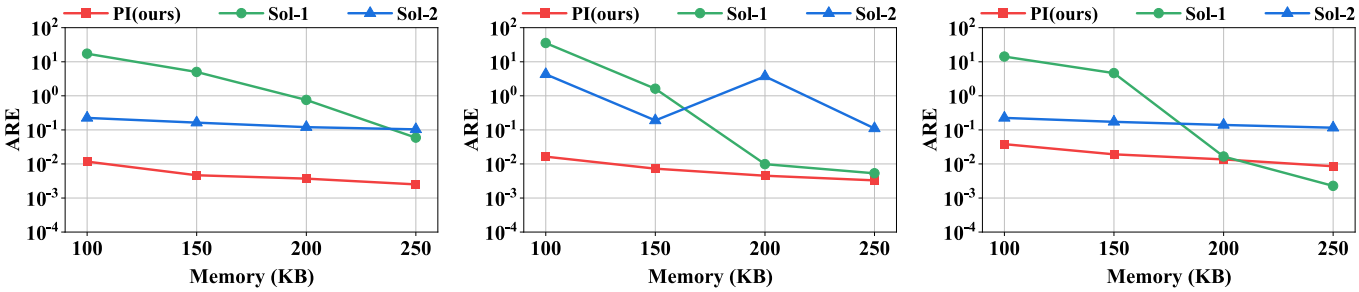
TABLE V
THE NUMBER AND PERCENTAGE OF PERSISTENT FLOWS AND FREQUENT FLOWS IN THE THREE REAL-WORLD DATASETS

Datasets	# Distinct Flows	# Persistent Flows	% Persistent Flows	# Frequent Flows	% Frequent Flows
CAIDA	327386	457	0.14%	2233	0.68%
MAWI	446578	300	0.07%	3039	0.68%
Network	317574	264	0.08%	100	0.03%



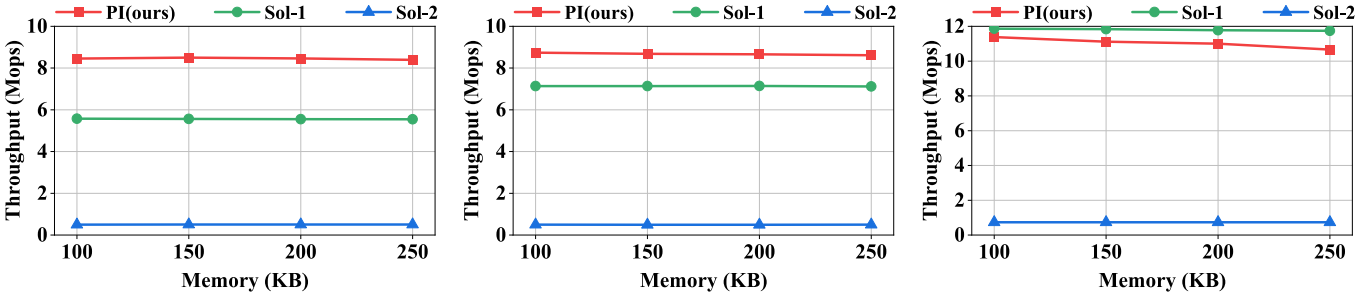
(a) CAIDA (b) MAWI (c) Network

Fig. 10. F1 Score on finding PI flows.



(a) CAIDA (b) MAWI (c) Network

Fig. 11. ARE on finding PI flows.



(a) CAIDA (b) MAWI (c) Network

Fig. 12. Throughput on finding PI flows.

peaks when $p = 4$ and $p = 5$. When the memory size is 100KB, the F1 Score peaks for $p = 7$, but the advantage is limited, as per Figure 5(a). In addition, we find that the ARE first decreases and then stabilizes with the increase of p, and the overall the throughput smoothly decreases as p increases, as per Figure 5(b). As shown in Figure 5(c), compared with $p = 4$, the throughput at $p = 5$ is acceptable, as it does not bring any major drawback. 2) Secondly, we compare the performance of the PISketch when p varies from 1 to 10 and with different N_{Total}, as shown in Figure 6(a)-6(c). We find that the trends of ARE and throughput for different N_{Total} are similar to those for different memory sizes, but the F1 Score of $N_{Total} = 8 \times 10^6$ is slightly smaller than that of $N_{Total} = 2 \times 10^6/5 \times 10^6$ when p is small. 3) Thirdly, we compare the performance of the PISketch when p varies from 1 to 10 and with different N_{PI}, as shown in Figure 7(a)-7(c). We find that the F1 Score decreases as N_{PI} increases, especially that of $N_{PI} = 1545$ is significantly smaller than that of $N_{PI} = 313/773$. In addition, ARE increases as N_{PI} increases, and the trend of throughput for

different N_{PI} is also similar to that of different memory sizes. For sub-experiments 2) and 3), we also find that each metric for $p = 5$ performs well or acceptably at their respective different N_{Total} or N_{PI}. In summary, we set p to 5 in our experiments.

Effects of N_{Total} (Figure 8(a)-8(c)): We find that the optimal values for N_{Total} are 5×10^6 and 7×10^6. In this experiment, we compare the performance of PISketch when N_{Total} varies from 1×10^6 to 8×10^6 with a step size of 1×10^6. When the memory size is 200/300KB, the F1 Score peaks when $N_{Total} = 7 \times 10^6$. When the memory size is 100KB, the F1 Score peaks when $N_{Total} = 5 \times 10^6$, as shown in Figure 8(a). In addition, the performance of PISketch's ARE and throughput at $N_{Total} = 5 \times 10^6$ is similar to the one at $N_{Total} = 7 \times 10^6$ as shown in Figures 8(b)-8(c). Thus, the optimal value of N_{Total} is 5×10^6 and 7×10^6, and we set $N_{Total} = 5 \times 10^6$.

Effects of N_{PI} (Figure 9(a)-9(c)): We find that the value of N_{PI} affects the F1 Score and ARE of PISketch. In this experiment, we compare the performance of PISketch when N_{PI} varies from 223 to 1545, where all variable values are 223, 313, 457, 773, 1545, in that order. We find that the F1 Score of PISketch first increases slightly and then gradually decreases with the increase of N_{PI}, ARE gradually increases with the increase of N_{PI}, and there is no obvious trend for throughput to change with N_{PI}.

Analysis: 1) The value of p should be selected based on a trade-off between the F1 Score and throughput, especially when its value is relatively small. 2) The performance under different scales of N_{Total} is similar, and users should pay more attention to the impact of different memory sizes under the same N_{Total} on the performance of PISketch. 3) When the memory size is fixed and N_{PI} exceeds a certain scale, PISketch needs to store too many PI flows, resulting in a slight decrease in accuracy and a slight increase in ARE.

D. Experiments on Finding PI Flows

We compare PISketch with two strawman solutions: 1) On-Off + CM sketch; 2) PIE + CM sketch. For PISketch and On-Off + CM sketch, we set the memory size range to 100KB-250KB. For PIE + CM sketch, the memory size range is set to 10000KB-25000KB. This means its memory range is 100 times the one of PISketch (applicable to Sections VI-D - VI-F). Specifically, we use PIE [3]/On-Off [2] to estimate flow persistency (*i.e.*, the time window number), and the CM sketch [21] to estimate flow frequency. We then combine them together to get the estimated persistency and infrequency, and finally find PI flows. The parameter configurations of PISketch and two strawman solutions is detailed in our supplemental material [59]. In the following, we refer to On-Off + CM sketch as **Sol-1** and PIE + CM sketch as **Sol-2** for short.

F1 Score (Figure 10(a)-10(c)): We find that the F1 Score of PISketch is much higher than the one of Sol-1 and Sol-2. On the three real-world datasets, the F1 Score of PISketch is around 22.1% and 57.6% higher than the one of Sol-1 and Sol-2 on average, respectively.

ARE (Figure 11(a)-11(c)): We find that the ARE of PISketch is significantly lower than the one of Sol-1 and Sol-2. On the three real-world datasets, the ARE of PISketch is around 820.9 (up to 1188.8) and 126.2 (up to 265.6) times lower than the one of Sol-1 and Sol-2 on average, respectively.

Throughput (Figure 12(a)-12(c)): We find that the insertion throughput of PISketch is higher than the one of Sol-1 and is obviously higher than the one of Sol-2. On the three real-world datasets, the throughput of PISketch is around 1.23 and 16.5 times higher than the one of Sol-1 and Sol-2 on average, respectively.

Analysis: Our results show that PISketch has better performance than Sol-1 and Sol-2, as expected. The main reasons are: 1) PISketch has converted frequencies and persistencies into weights. Therefore, there is no need to store them in each time window; 2) PISketch filters out most low-weight flows and finds PI flows more effectively through a competition (*i.e.*, eviction and replacement) mechanism. Also, the space complexity of PISketch is lower than the one of Sol-1, and much smaller than the one of Sol-2.

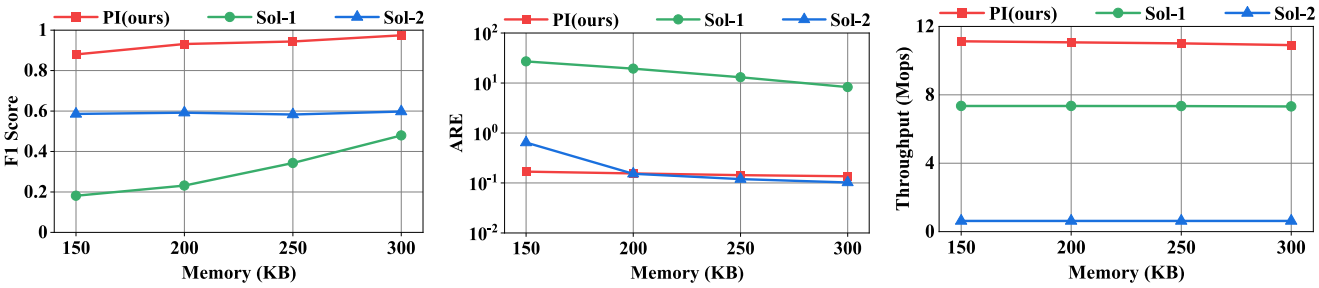
E. End-to-End Experiment I: Application in Preliminary APT Detection

Methodology: The implementation method of this experiment is similar to Section VI-D. Specifically, we run PISketch on the Synthetic Dataset I (see Section VI-A), and the output PI flows is the suspected APT flows. We use F1 Score, ARE, and throughput as evaluation metrics. Note that the memory range for PISketch and On-Off + CM sketch in Section VI-E-VI-F is set to 150KB-300KB, while for PIE + CM sketch is 15000KB-30000KB.

Experimental Results (Figures 13(a)-13(c)): We find that PISketch performs better than Sol-1 and Sol-2 in the preliminary screening of suspected APT flows. The results are as follows. 1) The F1 Score of PISketch is around 62.4% and 34.3% higher than the one of Sol-1 and Sol-2, respectively. 2) The ARE of PISketch is around 112.1 and 1.69 times lower than the one of Sol-1 and Sol-2 on average, respectively. 3) The throughput of PISketch is around 1.50 and 17.7 times higher than the one of Sol-1 and Sol-2 on average, respectively.

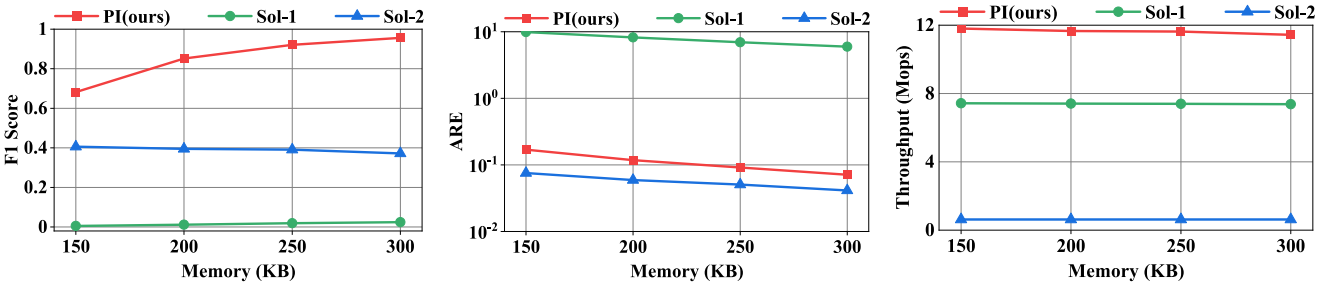
F. End-to-End Experiment II: Application in Preliminary FRP Detection

Methodology: We deploy FRP in two cloud servers. One server is hidden in the enterprise network and it deploys a FRP client (FRPc). The other server runs as a FRP server (FRPs), which can expose the server with FRP client to the Internet. When FRPs starts the service, FRPc connects through IP and port number, and they communicate every once in a while. Next, we use tcpdump [60] to capture the communication traffic between the two servers. Within the NAT, the IP address of FRPc may change due to DHCP. We repeat the experiment 20 times including setting up new cloud servers and capturing the communication traffic between FRPc and FRPs. Finally, we mix these captured traffic (40 FRP flows in total) into the normal flows to generate the Synthetic Dataset II described in Section VI-A, and evaluate whether PISketch can find the FRP



(a) F1 Score (b) ARE (c) Throughput

Fig. 13. Evaluation on APT preliminary detection.



(a) F1 Score (b) ARE (c) Throughput

Fig. 14. Evaluation on FRP preliminary detection.

flows out (similar to the experiment in Section VI-E). We still use F1 Score, ARE, and throughput as evaluation metrics.

Experimental Results (Figure 14(a)-14(c)): We find that PISketch outperforms than Sol-1 and Sol-2 on preliminary detection of suspected FRP flows. The results are as follows. 1) The F1 Score of PISketch is around 83.8% and 46.2% higher than the one of Sol-1 and Sol-2, respectively. 2) The ARE of PISketch is around 69.0 times lower than Sol-1 and 1.99 times higher than Sol-2, while Sol-2 uses 100 times the memory size. 3) The throughput of PISketch is around 1.57 and 18.7 times higher than the one of Sol-1 and Sol-2 on average, respectively.

G. Discussion of Section VI-E-VI-F

We have implemented two use cases for PISketch introduced in Section I-A: the preliminary detection of Advanced Persistent Threats (APT) [14], [15] and Fast Reverse Proxy (FRP) [16], but the applications of PISketch is not limited to them. The main contribution of PISketch in these two cases is the obvious efficiency improvement: it outputs only suspicious APT/FRP flows at a high speed with a small cost. In this way, users can further analyze these suspicious flows using existing solutions ($e.g.$, various intrusion detection systems (IDS)), which are **orthogonal** to PISketch. As future work, we plan to implement system-level deployment of PISketch in existing IDS solutions.

H. Examples of Problem Definition Parameter Settings

In this section, we present examples of the choice of three problem definition parameters ($i.e.$, the initial value L, the number of windows V, and the threshold $\mathcal{T}$) on different datasets (CAIDA, MAWI, Network). To achieve this goal, we have conducted extensive experiments on the aforementioned datasets. As shown in Figure 15, each line in these figures represents the following equation:

$$V(L+1)-frequency=\mathcal{T} \tag{13}$$

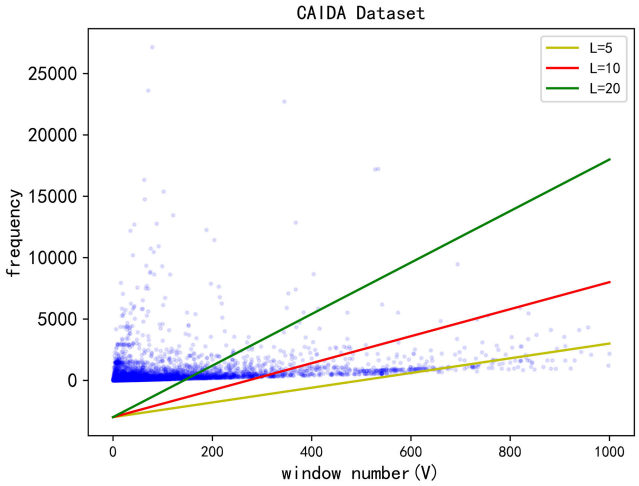
where *frequency* represents the frequency of flows in these datasets.

Conclusions: 1) The flows in the direction of the lower right corner of each line is the PI flows obtained under the corresponding parameters (L, V, $\mathcal{T}$). If a flow is closer to the lower right corner from the specified line, the more persistent and infrequent (PI) it is (and of course the higher the corresponding weight). 2) The lines of different colors in the figures represent the degree of "preference" for different parameters. Therefore, according to our continuous testing, we finally select the parameters corresponding to the red line in the figures as follows: $L=10$, $V=1000$ and $\mathcal{T}=3000$. Users can refer to our example and select these parameters flexibly according to their own needs.

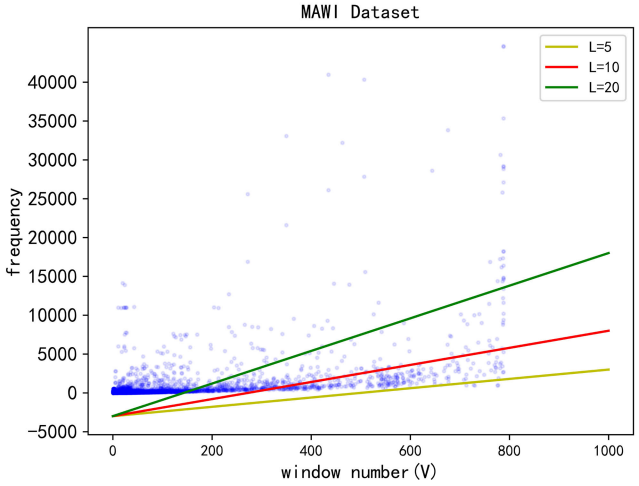
I. An Open Question

Question: Can the expected number of PI flows be asserted by the flow distribution?

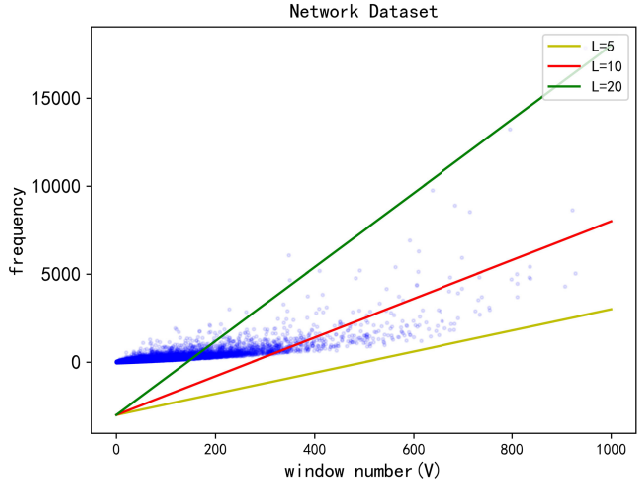
Answer: On the surface, flow distribution can only describe frequency information, but cannot describe persistence information. However, we may be able to estimate an upper limit for the expected number of PI flows through flow distribution: those with extremely low frequencies are definitely not PI flows, and those with extremely high frequencies are also not PI flows.



(a) CAIDA



(b) MAWI



(c) Network

Fig. 15. An example of problem definition parameter settings for finding PI flows. When $V = 0$, the absolute value of frequency is $\mathcal{T}$.

VII. Conclusion

In this paper, we propose PISketch, which is the first algorithm for finding PI (persistent and infrequent) flows in real time. We implement PISketch entirely on P4, FPGA, and conduct extensive experiments on CPU. We compare PISketch with two strawman solutions, one being On-Off + CM sketch and the other PIE + CM sketch. Our experimental results illustrate the advantages of our approach: PISketch can achieve around 22.1%/57.6% higher F1 Score, 1.23/16.5 times higher throughput, and 820.9/126.2 times lower ARE. Furthermore, our two end-to-end experiments demonstrate the good performance of PISketch in preliminary detection of APT and FRP flows.

Acknowledgment

The authors would like to thank their editor, and the anonymous reviewers for their thoughtful feedback.

References

[1] Z. Fan et al., "PISketch: Finding persistent and infrequent flows," in *Proc. ACM SIGCOMM Workshop Formal Found. Secur. Program. Netw. Infrastructures*, Aug. 2022, pp. 8–14.

[2] Y. Zhang et al., "On-off sketch: A fast and accurate sketch on persistence," *Proc. VLDB Endowment*, vol. 14, no. 2, pp. 128–140, Oct. 2020.

[3] H. Dai et al., "Identifying and estimating persistent items in data streams," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2429–2442, Dec. 2018.

[4] H. Dai, M. Li, A. X. Liu, J. Zheng, and G. Chen, "Finding persistent items in distributed datasets," *IEEE/ACM Trans. Netw.*, vol. 28, no. 1, pp. 1–14, Feb. 2020.

[5] H. Huang et al., "You can drop but you can't hide: K-persistent spread estimation in high-speed networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 1889–1897.

[6] H. Huang et al., "An efficient K-persistent spread estimator for traffic measurement in high-speed networks," *IEEE/ACM Trans. Netw.*, vol. 28, no. 4, pp. 1463–1476, Aug. 2020.

[7] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 101–114.

[8] Q. Huang et al., "SketchVisor: Robust network measurement for software packet processing," in *Proc. ACM SIGCOMM*, Aug. 2017, pp. 113–126.

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

[10] T. Yang et al., "HeavyKeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Trans. Netw.*, vol. 27, no. 5, pp. 1845–1858, Aug. 2019.

[11] Z. Liu et al., "NitroSketch: Robust and general sketch-based monitoring in software switches," in *Proc. SIGCOMM*, Aug. 2019, pp. 334–350.

[12] Y. Zhang et al., "CocoSketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proc. SIGCOMM*, Aug. 2021, pp. 207–222.

[13] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "SketchLib: Enabling efficient sketch-based monitoring on programmable switches," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, Mar. 2022, pp. 743–759.

[14] E. Cole, *Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization*. Waltham, MA, USA: Syngress, 2012.

[15] A. Alshamrani, S. Myneni, A. Chowdhary, and D. Huang, "A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1851–1877, 2nd Quart., 2019.

[16] (2021). *Fatedier/FRP: A Fast Reverse Proxy to Help you Expose a Local Server Behind a NAT or Firewall to the internet*. [Online]. Available: https://github.com/fatedier/frp

[17] L. Wang, K. S. Park, R. Pang, V. Pai, and L. Peterson, "Reliability and security in the CoDeeN content distribution network," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, Jun. 2004, pp. 171–184.

[18] M. S. Rahman, M. Y. S. Uddin, T. Hasan, M. S. Rahman, and M. Kaykobad, "Using adaptive heartbeat rate on long-lived TCP connections," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 203–216, Feb. 2018.

[19] A. Argyriou and V. Madisetti, "Using a new protocol to enhance path reliability and realize load balancing in mobile ad hoc networks," *Ad Hoc Netw.*, vol. 4, no. 1, pp. 60–74, Jan. 2006.

[20] I. Baldine, G. N. Rouskas, H. G. Perros, and D. Stevenson, "Jump-Start: A just-in-time signaling architecture for WDM burst-switched networks," *IEEE Commun. Mag.*, vol. 40, no. 2, pp. 82–89, Feb. 2002.

[21] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[22] M. Alizadeh et al., "pFabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 435–446, Aug. 2013.

[23] Y. Zhu et al., "Packet-level telemetry in large datacenter networks," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 479–491.

[24] A. Kumar, J. Xu, and J. Wang, "Space-code Bloom filter for efficient per-flow traffic measurement," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 12, pp. 2327–2339, Dec. 2006.

[25] T. Yang et al., "Guarantee IP lookup performance with FIB explosion," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 39–50.

[26] (2022). *Source Code and More details related to PISketch*. [Online]. Available: https://github.com/pkufzc/PISketch

[27] B. Lahiri, J. Chandrashekar, and S. Tirthapura, "Space-efficient tracking of persistent items in a massive data stream," in *Proc. 5th ACM Int. Conf. Distrib. Event-Based Syst.*, Jul. 2011, pp. 255–266.

[28] 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, pp. 1–29, Jun. 2017.

[29] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? Efficient set reconciliation without prior context," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 218–229, Aug. 2011.

[30] M. T. Goodrich and M. Mitzenmacher, "Invertible Bloom lookup tables," in *Proc. 49th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2011, pp. 792–799.

[31] A. Shokrollahi and M. Luby, "Raptor codes," *Found. Trends Commun. Inf. Theory*, vol. 6, nos. 3–4, pp. 213–322, May 2011.

[32] R. Schweller et al., "Reversible sketches: Enabling monitoring and analysis over high-speed data streams," *IEEE/ACM Trans. Netw.*, vol. 15, no. 5, pp. 1059–1072, Oct. 2007.

[33] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: Methods, evaluation, and applications," in *Proc. ACM SIGCOMM Conf. Internet Meas. (IMC)*, Oct. 2003, pp. 234–247.

[34] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.

[35] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. EATCS ICALP*, Jul. 2002, pp. 693–703.

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

[37] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 1449–1463.

[38] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, Aug. 2017.

[39] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "Heavy-Guardian: Separate and guard hot items in data streams," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 2584–2593.

[40] Y. Zhou et al., "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proc. SIGMOD*, Jun. 2018, pp. 741–756.

[41] Y. Zhao et al., "LightGuardian: A full-visibility, lightweight, in-band telemetry system using sketchlets," in *Proc. NSDI*, Apr. 2021, pp. 991–1010.

[42] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[43] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[44] S. Cohen and Y. Matias, "Spectral Bloom filters," in *Proc. ACM SIGMOD*, Jun. 2003, pp. 241–252.

[45] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic Bloom filters," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 120–133, Jan. 2010.

[46] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," *IEEE/ACM Trans. Netw.*, vol. 22, no. 4, pp. 1092–1105, Aug. 2014.

[47] Y. Wu et al., "Elastic Bloom filter: Deletable and expandable filter using elastic fingerprints," *IEEE Trans. Comput.*, vol. 71, no. 4, pp. 984–991, Apr. 2022.

[48] (2022). *Barefoot Tofino: World's Fastest P4-Programmable Ethernet Switch Asics*. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[49] (2020). *P4-16 Language Specification*. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-checksums

[50] (1997). *Bob Jenkins' Hash Function Web Page, Paper Published in Dr Dobb'S Journal*. [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html

[51] (2018). *The CAIDA Anonymized Internet Traces*. [Online]. Available: https://www.caida.org/catalog/datasets/overview/

[52] (2010). *MAWI Working Group Traffic Archive*. [Online]. Available: http://mawi.wide.ad.jp/mawi/

[53] (2014). *The Network Dataset Internet Traces*. [Online]. Available: http://snap.stanford.edu/data/

[54] L. Shang, D. Guo, Y. Ji, and Q. Li, "Discovering unknown advanced persistent threat using shared features mined by neural networks," *Comput. Netw.*, vol. 189, Apr. 2021, Art. no. 107937.

[55] J. Lu, K. Chen, Z. Zhuo, and X. Zhang, "A temporal correlation and traffic analysis approach for APT attacks detection," *Cluster Comput.*, vol. 22, no. S3, pp. 7347–7358, May 2019.

[56] J. Tan and J. Wang, "Detecting advanced persistent threats based on entropy and support vector machine," in *Proc. 18th Int. Conf. Algorithms Architectures Parallel Process. (ICAPP)*, Nov. 2018, pp. 153–165.

[57] D. Shick and A. Horneman, "Investigating advanced persistent threat 1 (APT1)," Carnegie Mellon Univ., Softw. Eng. Inst., Hanscom AFB, MA, USA, Tech. Rep., CMU/SEI-2014-TR-001, May 2014. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/Investigating_Advanced_Persistent_Threat_1_APT1_/6574880/files/12061442.pdf

[58] (2013). *Mila Parkour Contagio Malware Data-Base*. [Online]. Available: https://www.mediafire.com/folder/c2az029ch6cke/TRAFFIC_PATTERNS_COLLECTION#734479hwy1b97

[59] (2022). *The Supplementary Material of PISketch*. [Online]. Available: https://github.com/pkufzc/PISketch/blob/main/PISketch_Supplementary_Material.pdf

[60] (2018). *Tcpdump Examples*. [Online]. Available: https://hackertarget.com/tcpdump-examples/

Zhuochen Fan received the Ph.D. degree in computer science from Peking University in 2023, advised by Tong Yang. He is currently working as a Boya Post-Doctoral Fellow with the School of Computer Science, Peking University. He published papers in IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, ICDE, RTSS, ICPP, and ICNP. His research interests include data stream processing and algorithms, computer networks, and network measurements.

Zhoujing Hu is currently pursuing the bachelor's degree in computer science with Peking University. His research interests include data sketches and data stream processing systems.

Yuhan Wu received the bachelor's degree from the Department of Electrical Engineering and Computer Science, Peking University, in 2021. He is currently pursuing the Ph.D. degree in computer science with the School of Computer Science, Peking University, advised by Tong Yang. His research interests include computer networks and database, including key-value stores, network measurement, and sketches.

Jiarui Guo is currently pursuing the bachelor's degree in computer science with Peking University, advised by Tong Yang. His research interests include approximation algorithms in data streams and computer network systems.

Sha Wang is currently pursuing the M.S. degree with the College of Computer, National University of Defense Technology. Her main research interests include time sensitive networks.

Wenrui Liu is currently pursuing the bachelor's degree in computer science with Peking University, advised by Tong Yang. His research interests include network measurements, programmable switch, and network systems.

Tong Yang (Member, IEEE) received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently an Associate Professor with the School of Computer Science, Peking University. His research interests include network measurements, sketches, IP lookups, Bloom filters, and KV stores. He has served as a TPC Member for several premier conferences, such as INFOCOM and ICNP. He is currently an Associate Editor of *Knowledge and Information Systems*. He published dozens of papers in IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, *VLDB Journal*, SIGCOMM, NSDI, INFOCOM, SIGKDD, SIGMOD, VLDB, and ICDE.

Yaofeng Tu was born in 1972. He received the Ph.D. degree. He is currently a Researcher with the ZTE Nanjing Research and Development Center. His main research interests include big data, distributed systems, and machine learning.

Steve Uhlig received the Ph.D. degree in applied sciences from the Catholic University of Louvain, Belgium, in 2004. From 2004 to 2006, he was a Post-Doctoral Fellow with the Belgian National Fund for Scientific Research (F.N.R.S.). His thesis won the annual IBM Belgium/F.N.R.S. Computer Science Prize in 2005. From 2004 to 2006, he was a Visiting Scientist with Intel Research Cambridge, U.K., and the Department of Applied Mathematics, The University of Adelaide, Australia. From 2006 to 2008, he was with the Delft University of Technology, The Netherlands. From 2012 to 2016, he was a Guest Professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Since January 2012, he has been a Professor of networks and the Head of the Networks Research Group, Queen Mary University of London. Prior to joining the Queen Mary University of London, he was a Senior Research Scientist with Technische Universität Berlin/Deutsche Telekom Laboratories, Berlin, Germany.