# PISketch: Finding Persistent and Infrequent Flows

Zhuochen Fan*, Zhoujing Hu*, Yuhan Wu*, Jiarui Guo*, Wenrui Liu*,
Tong Yang*, Hengrui Wang*, Yifei Xu†, Steve Uhlig‡, Yaofeng Tu§
*Peking University, †University of California, Los Angeles,
‡Queen Mary University of London, §ZTE Corporation

## ABSTRACT

[1]Finding persistent and inactive 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 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.

## CCS CONCEPTS

• **Networks** → **Network measurement**.

## KEYWORDS

Persistent Items; Infrequent Items; Sketch; Weight Fusion;

*School of Computer Science, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China
†Computer Science Department, University of California, Los Angeles, USA
‡School of Electronic Engineering and Computer Science, Queen Mary University of London, U.K.
§ZTE Corporation, China
[1]Zhuochen Fan, Zhoujing Hu, and Yuhan Wu contribute equally to this paper. Corresponding author: Yaofeng Tu (tu.yaofeng@zte.com.cn) and Tong Yang (yangtongemail@gmail.com).

*FFSPIN '22, August 22, 2022, Amsterdam, Netherlands*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9329-4/22/08…$15.00
https://doi.org/10.1145/3528082.3544834

**ACM Reference Format:**
Zhuochen Fan, Zhoujing Hu, Yuhan Wu, Jiarui Guo, Wenrui Liu, Tong Yang, Hengrui Wang, Yifei Xu, Steve Uhlig, Yaofeng Tu. 2022. PISketch: Finding Persistent and Infrequent Flows. In *ACM SIGCOMM 2022 Workshop on Formal Foundations and Security of Programmable Network Infrastructures (FFSPIN '22), August 22, 2022, Amsterdam, Netherlands.* ACM, Philadelphia, PA, USA, 7 pages. https://doi.org/10.1145/3528082.3544834

## 1 INTRODUCTION

### 1.1 Background and Motivation

Finding frequent flows and persistent[2] flows has been considered as two important tasks in approximate data stream processing and network measurement [1–11]. Differently, we find that finding persistent and infrequent flows in data streams is also important, for example to identify activities that are sustained but inactive. Let us describe two possible use cases for finding persistent and infrequent flows.

- **Case 1.** Attack Defense. Many cyber attacks like Advanced Persistent Threats (APT) [12, 13] 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)[3] [14] can expose local servers behind a Network Address Translation (NAT) or firewall to the Internet. These connections are characterized by persistence and infrequency: FRP persistently produces packets, but the number of produced packets is very limited.

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**.

### 1.2 Prior Art and Limitations

To find PI flows[4], 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 [1] and the PIE [2, 3]. The state-of-the-art algorithms for estimating flow frequencies are sketch-based algorithms like the Count-Min (CM) sketch [15]. 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 [16, 17] in real

[2]Time-based windows: The window size is defined as a fixed period of time. We define the persistency of flow $f$ as the number of time windows where $f$ occurs.
[3]FRP supports HTTP/HTTPS, TCP, UDP and many other protocols, and forwards requests to internal services through domain names.
[4]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.

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[5] SRAM (Static RAM) to achieve high speed, and the size of the on-chip SRAM is limited [7, 19]. In summary, *the problem of finding PI flows is new and existing solutions do not work. We aim to design an efficient algorithm to approach it.*

## 1.3 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, where the length of each flow ID is 4 bytes. 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. Its time complexity is $O(1)$, and its insertion 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 and 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 3.1.

**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. The stayed flows are like becoming the presidential candidates, which 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 3.2 - 3.3 for details).

[5]On-chip memory, such as CPU cache and FPGA block RAM, *etc.* [18]

We implement PISketch entirely on P4 platform in Section 4. 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 5. We provide all the related code open-source at GitHub[6] [20].

**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 fully implement PISketch on P4 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.

## 2 RELATED WORK

## 2.1 Finding Persistent Flows

Several algorithms have been proposed to find persistent flows [1–4, 21, 22]. The state-of-art algorithms are On-Off [1] and PIE [2, 3]. 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 [23, 24] and a Raptor code [25] 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.

## 2.2 Frequency Estimation

Frequency (flow size) estimation consists in estimating the number of occurrences of flows. It has been widely studied. Sketches have proved their superiority in frequency estimation [9, 10, 26, 27]. Actually, they can achieve high accuracy and speed with limited memory usage. There are many sketch-based algorithms for estimating flow frequency [5, 7, 8, 15, 28–34], the most typical being the widely used Count-Min (CM) sketch [15]. 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

[6]https://github.com/pkufzc/PISketch

be selected. Besides the CM sketch, other typical sketch-based algorithms include sketches of CU [28], C [29], CSM [30], and ASketch [31], PyramidSketch [32], HeavyKeeper [8], HeavyGuardian [33], Cold Filter [34], *etc*. which focus on accurately estimating large/elephant flows. State-of-the-art sketch-based network measurement systems include SketchVisor [6], UnivMon [5], ElasticSketch [7], Nitrosketch [9], CocoSketch [10], LightGuardian [35], *etc*.

## 3 PISKETCH DESIGN

In this section, we first define the weight of a flow. Then, we introduce the data structure of PISketch. Next, we give the details of how to process incoming flows based on the weight.

### 3.1 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 $W_i$ $\left(1 \le i \le V, i \in \mathbb{Z}^+\right)$ of flow $f$ that occurs $O_i$ $\left(O_i \in \mathbb{Z}^+\right)$ 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 $W_i = 0$. $L$ $\left(L \in \mathbb{Z}^+\right)$ 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.

### 3.2 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 weights.

The data structure of the first part is a Bloom filter [36]. 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 the next part for weight calculation. A Bloom

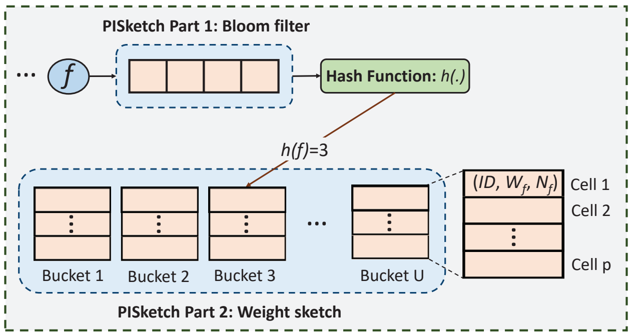


**Figure 1: Data structure of PISketch.**

filter[7] 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.

### 3.3 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: initialise 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

[7] A 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.

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.

**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.

## 4 P4 IMPLEMENTATION

We have fully built a P4 prototype of the PISketch on the Tofino switch [37]. In the P4 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. We list the utilization of various hardware resources on the switch in Table 1. 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.

**Table 1: 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% |

## 5 EXPERIMENTAL RESULTS

In this section, we show the experimental results of PISketch on CPU. First, we describe the experimental setup and metrics in Section 5.1 and Section 5.2, respectively. Next, we evaluate the performance of PISketch on different datasets and compare it with two strawman solutions in Section 5.3. Finally, through two end-to-end experiments, in Section 5.4 and Section 5.5, we provide two concrete applications of PISketch.

### 5.1 Experiment Setup

**Implementation:** We implement our algorithm and related algorithms in C++. Our hash function is the Bob Hash [38]. **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 one synthetic dataset. Each dataset contains about 5M flows.

**(1) CAIDA Dataset:** This IP Trace Dataset is streams of anonymized IP traces collected in 2018 by CAIDA [39].

**(2) MAWI Dataset:** This real packet traffic trace dataset is provided by the MAWI Working Group [40].

**(3) Network Dataset:** This dataset contains users' posting history on the stack exchange website [41]. 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.

**(4) Synthetic Dataset I:** Since large-scale real APT flows are too difficult to obtain, we synthesize this dataset by referring to some literature [42–45] 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 [46] 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 5.5). In this dataset, about 40 captured FRP flows are mixed with millions of normal flows.

### 5.2 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.

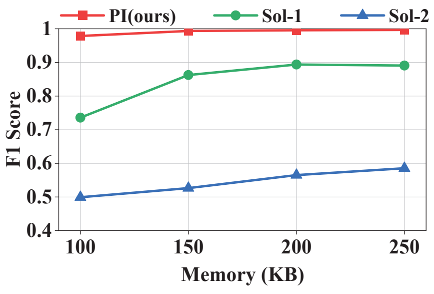
**(2) Average Relative Error (ARE):** Let $\hat{N_1}, \hat{N_2}, \ldots, \hat{N_k}$ be the estimated window number of the reported flows, and let $N_1, N_2, \ldots, N_k$ be the true window number of the reported flows. ARE is defined as $\frac{1}{k} \cdot \sum_{j=1}^{k} \frac{|N_j - \hat{N_j}|}{N_j}$.

**(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.

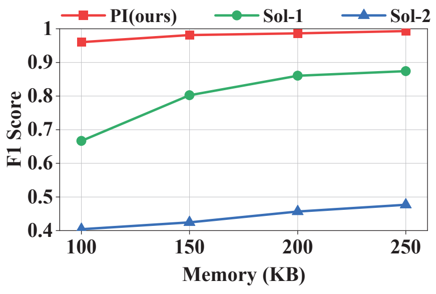
### 5.3 Experiments on Finding PI Flows

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.**[8] **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 5.3 - 5.5).** Specifically, we use PIE [2]/On-Off [1] to estimate flow persistency (*i.e.*, the time window number), and the CM sketch [15] 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

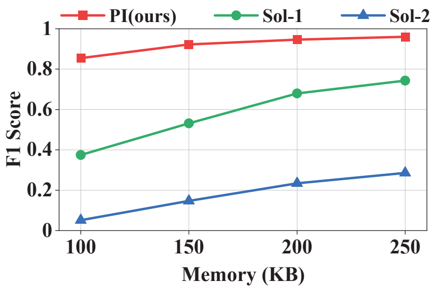
[8]The memory range for PISketch and On-Off + CM sketch in Section 5.5 is set to 150KB-300KB, while for PIE + CM sketch is 15000KB-30000KB.



(a) CAIDA

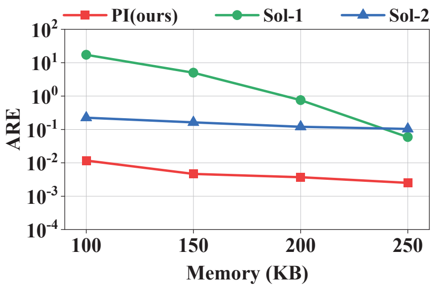


(b) MAWI

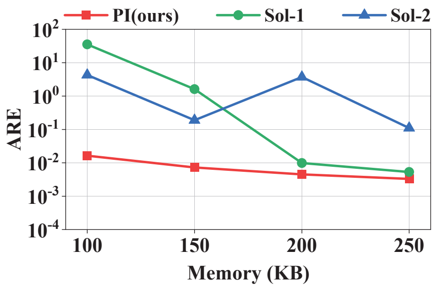


(c) Network

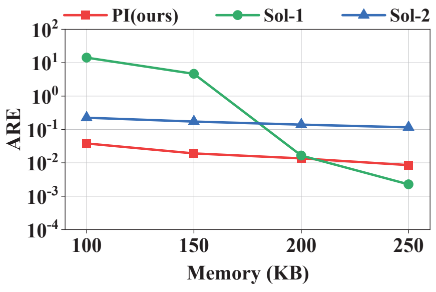
**Figure 2: F1 Score on finding PI flows.**



(a) CAIDA

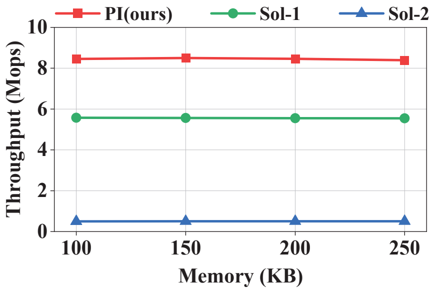


(b) MAWI

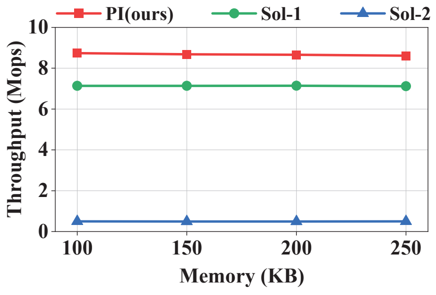


(c) Network

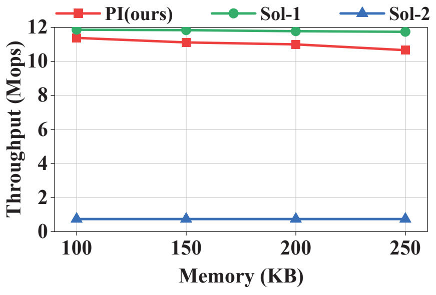
**Figure 3: ARE on finding PI flows.**



(a) CAIDA



(b) MAWI



(c) Network

**Figure 4: Throughput on finding PI flows.**

supplemental material [47]. In the following, we refer to On-Off + CM sketch as **Sol-1** and PIE + CM sketch as **Sol-2** for short.

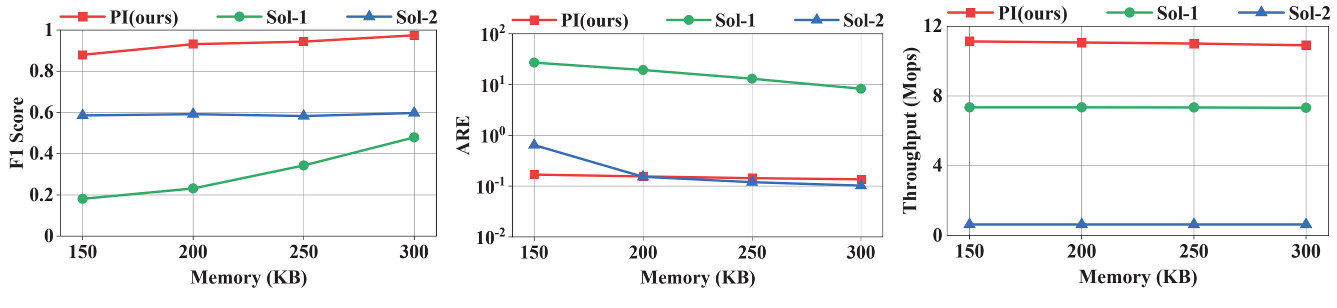
**Parameter Settings:** We set $L = 10$, $V = 1000$, $\mathcal{T} = 3000$, and $p = 5$.

**F1 Score (Figure 2(a)-2(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 3(a)-3(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.

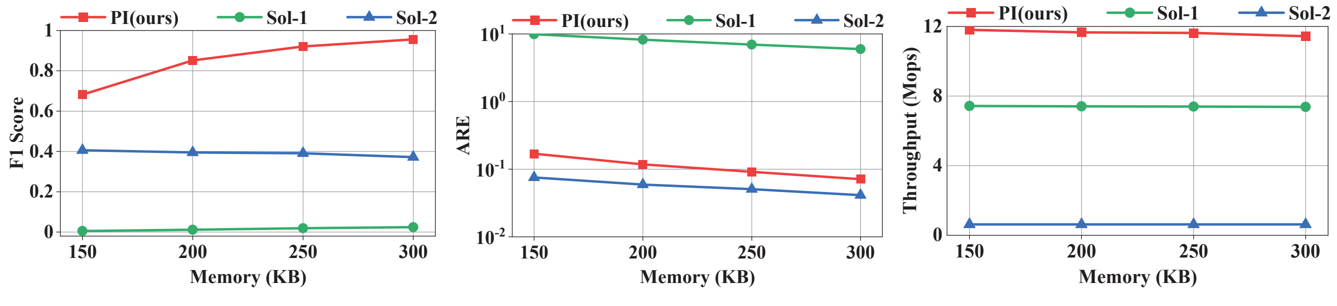
**Throughput (Figure 4(a)-4(c)):** *We find that the insertion throughput of PISketch is higher than the one of Sol-1 and is obviously higher than the one of Sol-2.* On the three real-world datasets, the throughput of PISketch is around 1.23 and 16.5 times higher than the one of Sol-1 and Sol-2 on average, 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.



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

**Figure 5: Evaluation on APT preliminary detection.**



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

**Figure 6: Evaluation on FRP preliminary detection.**

## 5.4 End-to-End Experiment I: Application in Preliminary APT Detection

**Methodology:** The implementation method of this experiment is similar to Section 5.3. Specifically, we run PISketch on the Synthetic Dataset I (see Section 5.1), and the output PI flows is the suspected APT flows. We use F1 Score, ARE, and throughput as evaluation metrics.

**Experimental Results (Figures 5(a)-5(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.

## 5.5 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 [48] 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 5.1, and evaluate whether PISketch can find the FRP flows out (similar to the experiment in Section 5.4). We still use F1 Score, ARE, and throughput as evaluation metrics.

**Experimental Results (Figure 6(a)-6(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 larger memory size than ours. 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.

## 6 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 and conduct extensive experiments on CPU. Specifically, 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

This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, National Natural Science Foundation of China (NSFC) (No. U20A20179, 61832001).

## REFERENCES

[1] Yinda Zhang, Jinyang Li, Yutian Lei, Tong Yang, Zhetao Li, Gong Zhang, and Bin Cui. On-off sketch: a fast and accurate sketch on persistence. *Proc. VLDB Endow.*, 14(2):128–140, 2021.
[2] Haipeng Dai, Muhammad Shahzad, Alex X. Liu, Meng Li, Yuankun Zhong, and Guihai Chen. Identifying and estimating persistent items in data streams. *IEEE/ACM Transactions on Networking*, 26(6):2429–2442, 2018.
[3] Haipeng Dai, Meng Li, Alex X. Liu, Jiaqi Zheng, and Guihai Chen. Finding persistent items in distributed datasets. *IEEE/ACM Transactions on Networking*, 28(1):1–14, 2020.
[4] He Huang, Yu-E Sun, Chaoyi Ma, Shigang Chen, You Zhou, Wenjian Yang, Shaojie Tang, Hongli Xu, and Yan Qiao. An efficient k-persistent spread estimator for traffic measurement in high-speed networks. *IEEE/ACM Transactions on Networking*, 28(4):1463–1476, 2020.
[5] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*, page 101–114, 2016.
[6] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*, pages 113–126, 2017.
[7] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*, pages 561–575, 2018.
[8] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. Heavykeeper: An accurate algorithm for finding top-$k$ elephant flows. *IEEE/ACM Transactions on Networking*, 27(5):1845–1858, 2019.
[9] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*, page 334–350, 2019.
[10] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM Conference (SIGCOMM '21)*, page 207–222, 2021.
[11] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. Sketchlib: Enabling efficient sketch-based monitoring on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 743–759, 2022.
[12] Eric Cole. *Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization.* Syngress Publishing, 2012.
[13] Adel Alshamrani, Sowmya Myneni, Ankur Chowdhary, and Dijiang Huang. A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities. *IEEE Communications Surveys and Tutorials*, 21(2):1851–1877, 2019.
[14] fatedier/frp: A fast reverse proxy to help you expose a local server behind a nat or firewall to the internet. https://github.com/fatedier/frp.
[15] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
[16] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the 2013 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '13)*, page 435–446, 2013.
[17] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, pages 479–491, 2015.
[18] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X. Liu, Qi Li, and Laurent Mathy. Guarantee ip lookup performance with fib explosion. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*, pages 39–50, 2014.
[19] Abhishek Kumar, Jun Xu, and Jia Wang. Space-code bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications*, 24(12):2327–2339, 2006.
[20] Source code and more details related to PISketch. https://github.com/pkufzc/PISketch.
[21] Bibudh Lahiri, Jaideep Chandrashekar, and Srikanta Tirthapura. Space-efficient tracking of persistent items in a massive data stream. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based System (DEBS '11)*, pages 255–266, 2011.
[22] You Zhou, Yian Zhou, Min Chen, and Shigang Chen. Persistent spread measurement for big network data based on register intersection. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):1–29, 2017.
[23] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. What's the difference?: efficient set reconciliation without prior context. *ACM SIGCOMM Computer Communication Review*, 41(4):218–229, 2011.
[24] Michael T. Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799, 2011.
[25] Amin Shokrollahi, Michael Luby, et al. Raptor codes. *Foundations and trends® in communications and information theory*, 6(3–4):213–322, 2011.
[26] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A Dinda, Ming-Yang Kao, and Gokhan Memik. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking*, 15(5):1059–1072, 2007.
[27] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement (IMC '03)*, pages 234–247, 2003.
[28] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *ACM SIGCOMM Computer Communication Review*, 32(4):323–336, 2002.
[29] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703, 2002.
[30] Tao Li, Shigang Chen, and Yibei Ling. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking*, 20(5):1622–1634, 2012.
[31] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 1449–1463, 2016.
[32] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.*, 10(11):1442–1453, 2017.
[33] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 2584–2593, 2018.
[34] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 741–756, 2018.
[35] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. Lightguardian: A Full-Visibility, lightweight, in-band telemetry system using sketchlets. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 991–1010, 2021.
[36] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[37] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/.
[38] Bob jenkins' hash function web page, paper published in dr dobb's journal. http://burtleburtle.net/bob/hash/evahash.html.
[39] The CAIDA Anonymized Internet Traces. https://www.caida.org/catalog/datasets/overview/.
[40] MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/.
[41] The Network dataset Internet Traces. http://snap.stanford.edu/data/.
[42] Longkang Shang, Dong Guo, Yuede Ji, and Qiang Li. Discovering unknown advanced persistent threat using shared features mined by neural networks. *Computer Networks*, 189:107937, 2021.
[43] Jiazhong Lu, Kai Chen, Zhongliu Zhuo, and XiaoSong Zhang. A temporal correlation and traffic analysis approach for apt attacks detection. *Cluster Computing*, 22(3):7347–7358, 2019.
[44] Jiayu Tan and Jian Wang. Detecting advanced persistent threats based on entropy and support vector machine. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 153–165, 2018.
[45] Deana Shick and Angela Horneman. Investigating advanced persistent threat 1 (apt1). *Technical Report CMU/SEI-2014-TR-001, Carnegie Mellon University*, 2014.
[46] Mila Parkour (2013) Contagio malware data-base. https://www.mediafire.com/folder/c2az029ch6cke/TRAFFIC_PATTERNS_COLLECTION#734479hwy1b97.
[47] The supplementary material of PISketch. https://github.com/pkufzc/PISketch/blob/main/PISketch_Supplementary_Material.pdf.
[48] Tcpdump Examples. https://hackertarget.com/tcpdump-examples/.