PSSketch: Finding Persistent and Sparse Flow with High Accuracy and Efficiency

Jiayao Wang
wangjiayao@nudt.edu.cn
National University of Defense Technology, Changsha, China

Qilong Shi
sql23@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Xiyan Liang
2212207@mail.nankai.edu.cn
Nankai University
Tianjin, China

Han Wang
wangh15@pcl.ac.cn
Peng Cheng Laboratory
Shenzhen, China

Wenjun Li*
wenjunli@pku.org.cn
Peng Cheng Laboratory
Shenzhen, China

Ziling Wei
weiziling@nudt.edu.cn
National University of Defense Technology, Changsha, China

Weizhe Zhang
wzzhang@hit.edu.cn
Harbin Institute of Technology
Harbin, China

Shuhui Chen*
shchen@nudt.edu.cn
National University of Defense Technology, Changsha, China

Abstract

Finding persistent sparse (PS) flow is critical to early warning of various threats. Previous works have predominantly focused on either heavy or persistent flows, with limited attention given to PS flows. Although some recent studies pay attention to PS flows, they struggle to establish an objective criterion due to insufficient data-driven observations, resulting in reduced accuracy. In this paper, we define a new criterion "anomaly boundary" to distinguish PS flows from regular flows. Specifically, a flow whose persistence exceeds a threshold will be protected, while a protected flow with a density lower than a threshold is reported as a PS flow. We then introduce PSSketch, a high-precision layered sketch, to find PS flows. PSSketch employs variable-length bitwise counters, where the first layer tracks the frequency and persistence of all flows, and the second layer protects potential PS flows and records overflow counts from the first layer. Some optimizations have also been implemented to reduce memory consumption further and improve accuracy. The experiments show that PSSketch reduces memory consumption by 1-2 orders of magnitude compared to the strawman solution combined with existing work. Compared with SOTA solutions for finding PS flows, it outperforms up to 2.94x higher in F1 score and reduces ARE by 1-2 orders of magnitude. Meanwhile, PSSketch achieves a higher throughput than these solutions.

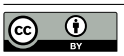
CCS Concepts

• **Information systems → Data stream mining**.

Keywords

Data streams; Data mining; Approximate algorithm; Sketch

*Corresponding authors.



KDD '25, Toronto, ON, Canada

ACM ISBN 979-8-4007-1454-2/2025/08
https://doi.org/10.1145/3711896.3737094

ACM Reference Format:
Jiayao Wang, Qilong Shi, Xiyan Liang, Han Wang, Wenjun Li, Ziling Wei, Weizhe Zhang, and Shuhui Chen. 2025. PSSketch: Finding Persistent and Sparse Flow with High Accuracy and Efficiency. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3711896.3737094

KDD Availability Link:
The source code of this paper has been made publicly available at https://doi.org/10.5281/zenodo.15494637.

1 Introduction

The increasing volume of data flow presents a growing challenge in ensuring network security through traffic analysis. Approximate flow processing algorithms have gained popularity due to the excessive resource consumption of precise analysis. Sketches are widely utilized as a standard data compression tool. In the existing literature, the task is often divided into spatial and temporal perspectives, where the number of occurrences and time span will be recorded. Heavy flow detection is a common task. By approximating the occurrence times of flows, this method identifies flows that appear significantly more frequently than others [6, 8, 10, 11, 33, 47, 49, 52, 55]. The time span is typically represented by dividing the time into equal intervals, referred to as time windows, and counting the number of windows in which a flow has appeared, known as flow persistence. Persistent flow detection has garnered increasing attention in recent years [7, 9, 15, 21, 51].

However, many attacks are not heavy flows; instead, they tend to conceal themselves and persist for a long time. For example, backdoor programs or reverse proxies may continuously send a few packets to the target to ensure stable control or retrieve information. Beyond the scope of attacks, detecting persistent sparse (PS) flows is also valuable for server administrators. Some users may access resources continuously over long periods, consuming significant server resources while contributing little in return. For instance, in online gaming, account sellers may create numerous low-activity accounts to collect daily rewards. Identifying these

behaviors typically requires high-cost technologies, such as regular expression matching and AI. Deploying these functionalities across all traffic is often impractical. Therefore, we need a method capable of preemptively detecting persistent sparse flows to alleviate their workload. Fortunately, many previous studies have demonstrated that the majority of flows exhibit low persistence, while long-duration flows are often associated with larger data volumes, such as file transfers and online videos. Consequently, identifying low-frequency persistent flows can significantly reduce the number of flows that require processing for threat detection, while also minimizing the interference from most normal flows.

Current frequency-focused algorithms primarily target heavy flows, rendering them incapable of capturing infrequent flows. Thus, these algorithms are not applicable in the aforementioned scenarios. Meanwhile, existing algorithms designed to track persistent flows do not consider frequency statistics. An intuitive strawman solution is to combine these two approaches. However, as shown in our experiments, without optimization for the PS task, such a solution exhibits low memory efficiency, severely limiting its practicality. PISketch [15] is currently the only model optimized for the PS task. However, the criterion it employs to filter PS flows is not based on data observations. As a result, the filtered flows are not consistently persistent and sparse. Furthermore, PISketch uses a conventional sketch to store flow information, which leads to low accuracy.

In this paper, we introduce a novel model, PSSketch, aimed at accurately and efficiently identifying PS flows. Our design is centered around the following key points:

- Objective Criterion. We analyze three commonly used datasets derived from real networks. The persistence and density distribution of flows exhibit similar cliff-like characteristics. Thus, we can draw an "anomaly boundary", using persistence and density to define PS flows.
- Protecting PS Flows. Traditional sketches store all flows within a single structure, where new flows often replace existing target flows (i.e., PS flows in our task) due to memory limitations. We propose a layered data structure, PSSketch, in which the first layer filters persistent flows, while the second layer stores PS candidates.
- Bit-Level Counters and Overflow Storage. Existing two-layer sketch structures often suffer from data redundancy, where information is stored twice. To address that, the first layer contains only small bit-level counters, while the second layer tracks overflow times, which improves memory utilization.
- Optimization. We raise a probabilistic replacement strategy to protect PS flows during their initial stages, along with a pruning mechanism to eliminate non-PS flows, thereby optimizing memory usage. Additionally, Single Instruction Multiple Data (SIMD) is also used to reduce memory access, significantly enhancing throughput.

Experiments show that, in comparison to the strawman solution, PSSketch achieves a reduction in memory requirements by 1 to 2 orders of magnitude. When compared to the SOTA PS flow-finding model, PISketch, we increase the F1 Score from a range of 0.4-0.6 to over 0.99. Additionally, the average relative error is reduced by 1 to 2 orders of magnitude. Furthermore, PSSketch outperforms both the strawman solution and PISketch in terms of throughput.

In Section 2, we introduce the background and related work. The observations and criteria we employed are detailed in Section 3. Next, we present the proposed PSSketch in Section 4, followed by a mathematical analysis in Section 5. Subsequently, PSSketch is evaluated in Section 6. Finally, Section 7 concludes our work.

2 Background and Related Work

2.1 Background

Before introducing our task, we will first provide some key definitions, and more detailed versions can be found in Appendix A.

Flow: Suppose E is an ID set of all possible incoming flows. To simplify the problem, we assume that each element contains only the ID of a flow, denoted as e. A data flow S is a multiset of N elements $\langle e_1, e_2, ..., e_N \rangle (e_i \in E)$.

Frequency: In a data flow S, the frequency of an element $e_i \in S$ refers to its multiplicity, denoted as f_i. That means e_i appears f_i times in the S.

Time Window: We divide a longer time interval into many small intervals of equal length t, each of these intervals is called a time window T_i.

Persistence: The number of time windows covered by packets from a certain flow e is denoted as its persistence p_i. This value will either increase by one or remain constant in each time window.

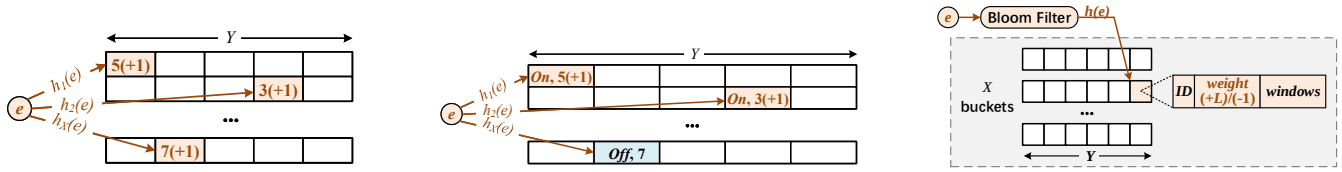
Density: On longer time spans, density can be calculated directly from the frequency f_e and persistence p_e as $d_e = \frac{f_e}{p_e}$.

Unlike packet classification [12, 24, 25, 27, 39–42, 54] and deep packet inspection [18, 19, 38, 43, 53], identifying specific data flows requires storing statistical information, such as frequency and persistence. However, by the end of the 20th century, the rapidly increasing volume of data rendered comprehensive storage impractical. To address this issue, probabilistic data structures such as Filters [7, 13, 14, 17] and Sketches [6, 10, 11, 20, 26, 33, 35–37, 44, 47–50, 52, 55] were introduced. These structures utilize hash functions to store information in a compact format, thereby reducing memory consumption at the expense of accuracy. The trade-off between accuracy and memory usage is central to this issue. In Section 2.2, we discuss how related works count frequency and persistence, with a new task proposed in recent years: finding flows with both high persistence and low frequency. Some other approaches [16, 22, 23, 28–30, 32, 34, 45, 46] have also been proposed for finding specific flows; however, due to their limited adoption, they will not be discussed in this paper.

2.2 Related Works

Frequency Estimation. Frequency estimation and persistence estimation are two main tasks in network data stream mining. Frequency estimation refers to counting the number of occurrences for each flow. Count-Min Sketch (CMSketch) [11] is one of the most widely applied models. To illustrate the core concept of frequency estimation, we use CMSketch as an example.

As shown in Figure 1(a), CMSketch consists of X buckets, each with Y counters. When a flow arrives, its ID is processed by X different hash functions, generating X indices $I_1, I_2, ..., I_x$. The corresponding counters at each index are then incremented. To query the frequency of a flow, the same process is performed, and it returns the minimum value among the X counters.



(a) CMSketch (b) On-off Sketch (c) PISketch

Figure 1: Structure of Related Works.

Due to memory constraints, most sketch-based frequency estimation solutions only focus on identifying "heavy flows". They target the few flows that occur with the highest frequency. Thus, they make the newly arriving flows replace those with low frequencies. So far, none of these models focus on counting infrequent flows, which makes it impossible for them to find PS flows.

Persistence Estimation. K-ary Sketch [21] is the first to incorporate the temporal dimension by comparing the frequency of flows across different time intervals to detect changes. BurstSketch [31] further extends this research within the Filter-Sketch framework. On-off Sketch (OOSketch) [51] explicitly introduces the concept of time windows, using persistence to determine whether a flow consistently appears over an extended period. It introduces an on-off switch based on CMSketch so that the same counter can only be modified once in a time window, as shown in Figure 1(b). These solutions do not account for the number of occurrences within each window, leaving them unsuitable for finding PS flows.

Finding PS flows. In our application scenario, PS flows are defined by their low frequency and high persistence. A straightforward approach is to combine existing models. We propose a *Strawman solution*, which uses CMSketch to track frequency and OOSketch to record persistence. However, it suffers from low memory efficiency and throughput. PISketch [15] is currently the only model specifically optimized for PS flows, as it integrates frequency and persistence. As illustrated in Figure 1(c), PISketch employs a filter to check whether a flow has appeared in the current time window and records a weight (W) in the sketch. If a flow appears for the first time in a time window, its weight is increased by a predefined value L ($L > 1$). Each subsequent occurrence of the flow within the same time window decreases the weight by one, and the flow is removed once W reaches zero. Although the weight indicates PS flow characteristics to some extent, it does not serve as an accurate criterion, as we will introduce in Section 3. Moreover, the sketch structure used by PISketch is relatively simple and lacks optimization based on data characteristics; therefore, it exhibits low memory utilization.

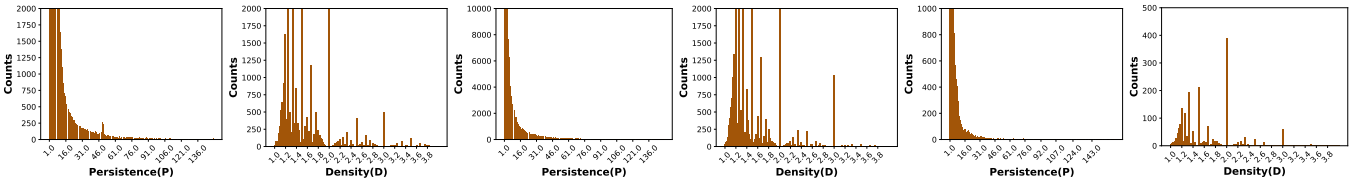
3 Observation and Criterion

In this section, we will define "What is PS flows", specifically the criteria for finding PS flows. Unlike traditional tasks, such as finding heavy flows, finding persistent flows, or cardinality estimation, filtering PS flows requires attention to both frequency and persistence as statistical metrics. Therefore, we first need to examine the distribution characteristics of these two metrics in common real-world datasets. Based on these observations, we will then determine how to use these distribution characteristics to provide a more objective and accurate definition, aligned with our research goals. We have studied three common real-world datasets: CAIDA [2], MAWI [3], and Campus [49]. Figure 2 shows their distribution characteristics. Intuitively, they exhibit similar patterns. Taking CAIDA as an example, most flows have persistence below 10, while flows with persistence above 50 account for only 3.141%. Regarding density, most flows exhibit values below 2, and the number of flows decreases as density increases. Conversely, as density approaches 1, the number of flows drops sharply, indicating that regular flows with high persistence rarely have densities near 1. The flows with a persistence more than 150 or density greater than 3.8 only account for a very small part, and therefore are not drawn in the figure.

Considering that our goal is to perform pre-filtering for high-cost threat detection techniques like DPI and AI, we need to define thresholds to determine the boundary for filtering. Fortunately, the distribution characteristics above exhibit abrupt changes, which can be leveraged to determine the boundaries. The idea for defining this boundary is closely related to our goal: Firstly, we focus only on persistent flows. A behavior usually requires a large amount of data to form; even if we input non-persistent flows into DPI or AI systems, it is difficult to get meaningful results. Additionally, flows with short durations account for a large proportion, hindering the reduction of the load on high-cost modules. Secondly, within the persistent flows, we focus on those with lower density, as our goal is to identify hidden threat behaviors—there are already many well-established methods for detecting high-density flows, such as DOS attacks. This implies that we need a two-dimensional threshold to define our target, which we refer to as the "anomaly boundary".

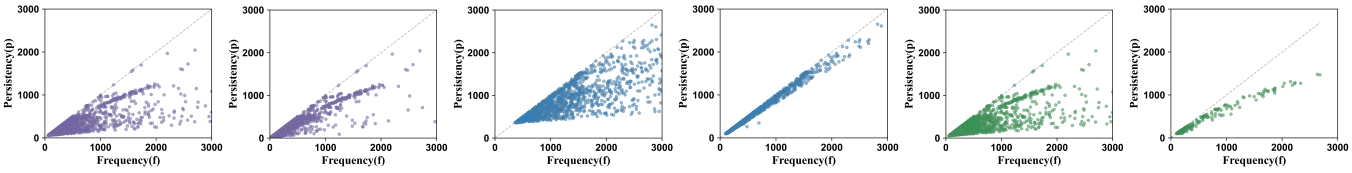
We define the "anomaly boundary" in this paper as follows: a flow is reported as a PS flow if its persistence exceeds the threshold p_0 and then its density falls below the threshold d_0(e.g., $p_0 \in [20, 60]$, $d_0 \in [1.1, 1.5]$). A more lenient boundary reduces the false negatives in threat warning but increases the workload. In practical deployment, the thresholds should be adjusted according to performance and security requirements. Based on the definition, we will detail our proposed PSSketch in Section 4. Before that, we compared the top 2,000 PS flows reported by PISketch with those reported by our method. Specifically, in PISketch, flows with higher weights W are ranked higher, while under our definition, persistent flows with lower density are ranked higher. Note that non-persistent flows will never be considered PS flows. Then, we traverse the three datasets to obtain the actual statistics of these reported flows.

Figure 3 demonstrates the results and shows the advantage of using "anomaly boundary" over PISketch's weights W. In all these datasets, we report PS flows that are much more concentrated near the line $l : f = p$. This means that we can preferentially find those sparse flows, while the flows found by PISketch are distributed over the entire possible range without discrimination.



(a) Persistence-CAIDA (b) Density-CAIDA (c) Persistence-Campus (d) Density-Campus (e) Persistence-MAWI (f) Density-MAWI

Figure 2: The Distribution Characteristics of Different Datasets.



(a) PISketch-CAIDA (b) Ours-CAIDA (c) PISketch-Campus (d) Ours-Campus (e) PISketch-MAWI (f) Ours-MAWI

Figure 3: Top-2K PS Flows Reported by PISketch and Our Method under Different Datasets.

4 Proposed PSSketch

4.1 Data Structure

As illustrated in Figure 4, PSSketch is composed of two layers: the Competition Layer (CL) and the Protection Layer (PL). The Competition Layer consists of X buckets. Each bucket contains Y entries. One entry is responsible for storing information about one flow, including the flow's fingerprint (FP), frequency (f), persistence (p), and flags for storing additional information. Thus, we can denote the j^{th} entry in the i^{th} bucket as $Bk[i].Et[j] = \{FP, f, p, flags\}$. The length of FP, counter f, and counter p can vary depending on the load, while the $flags$ are fixed at 2 bits, including a window identifier W and an overflow identifier OF. The flag W is initialized to 0 at the beginning of each time window and is set to 1 when the flow in the entry comes in that window. The flag OF indicates whether the flow has reported an overflow. We denote the two flags as $\boxed{10}$, that means the flow has already appeared in this window and its counter does not overflow. All flows enter the CL first. With the need to store a large amount of information, the counters are compressed to under 8 bits. Once the counter of a flow in the CL overflows, it is reported to the PL. The Protection Layer consists of an entry vector of length R. The content of an entry includes a flow's ID and the overflow times of its CL counters, that is, $Prt[i] = \{ID_e, f^e_{of}, p^e_{of}\}$. Since the persistent flows represent only a small portion of the total flows, the Protection Layer can store ID, which is longer than FP, to reduce hash collisions.

Recording overflow times greatly reduces the length of counters in two layers, avoiding information redundancy. In addition, the data transmission is unidirectional from CL to PL, saving the delay introduced by the data exchange. However, the above two problems are main bottlenecks of many traditional double-layer sketches.

4.2 Operations

4.2.1 Insert. First, we apply a hash function to a flow's ID and get its fingerprint $FP_e = h(e)$. The m^{th} bucket is selected where $m = FP_e \mod X$. In that bucket, there are three possible cases:

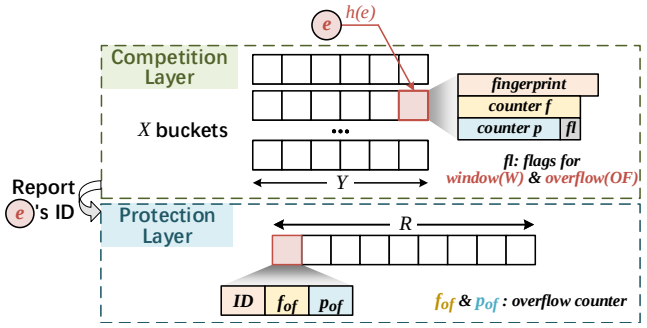


Figure 4: Structure of PSSketch.

Case 1: Search $BK[m]$ and find an entry that $BK[m].Et[n].FP = FP_e$. If it is found, it transitions to $Update$.

Case 2: If no such entry, we search for the first empty entry where $BK[m].Et[n].FP = 0$ and record $FP = FP_e, f = 1, p = 1, flag = \boxed{10}$. Therefore, e is added to $BK[m].Et[n]$ successfully.

Case 3: If none of the entries in $BK[m]$ satisfy $BK[m].Et[n].FP = FP_e$ and none are empty, then $BK[m]$ is full and transitions to $Contend$.

The pseudocode of $Insert$ can be found in our Github [4].

4.2.2 Update. When a flow e has been recorded in $BK[m].Et[n]$, we inspect the flag W. If $BK[m].Et[n].flag[W] = 0$, flow e has not appeared in the current time window. Thus, $BK[m].Et[n].p$ is incremented by 1. Moreover, $BK[m].Et[n].f$ is incremented by 1, and $BK[m].Et[n].flag[W]$ is updated to 1. Note that we will reset all $flag[W]$ to 0 at the beginning of each time window.

Since the counters used in the Competition Layer are small, f and p may soon overflow after being incremented. The two cases are:

Case 1: f overflows first. In our design, the width of counter f is 8 bits, while counter p is 6 bits. If f overflows first, it indicates that f_e has reached $2^8 = 256$, while p_e is at most $2^6 - 1 = 63$. In this case, the density $D = f/p > 4$. Flow e will not be reported in this case

because even if it meets the condition to become a persistent flow (i.e., $p_e > p_0$), its high density disqualifies it from being protected. Instead, we eliminate the flow.

Case 2: p overflows first. At this point, flow e meets the condition for persistence, and its density $D = f/p < 4$. The flow will be reported to the PL for protection. The $Flag[OF]$ is now set to 1, and $BK[m].Et[n].p$ is reset to 0.

Case 3: If the $Flag[OF]$ has already been 1, this flow is protected. Whether f or p overflows, the overflow is directly reported to the PL, and both $BK[m].Et[n].f$ and $BK[m].Et[n].p$ are reset.

Based on the cases above, the PL performs two types of updates:

Item Creation (Case 2): When flow e enters the PL for the first time, it searches for the first empty entry, denoted as $Prt[k]$. It writes $\{ID_e, 0, 1\}$ into it, representing the flow's identifier, counter f overflow times, and counter p overflow times. If there is no empty entry, we replace the flow with the largest density. Since persistent flows account for a small fraction of all flows, we can avoid this by appropriately increasing the length of the PL according to the load.

Item Update (Case 3): If flow e has already been protected, no matter f or p overflows, the Protection Layer finds the entry corresponding to the flow (also denoted as $Prt[k]$ here) and increments either f or p accordingly. Then, the *report* returns "SUCCESS".

The pseudocode of $Update$ can also be found on GitHub [4].

4.2.3 Contend. Contend occurs when the bucket $BK[m]$ is filled in CL. The newly inserted flow e will attempt to replace an existing flow in the bucket. Since, both the frequency f and persistence p of the new flow e are equal to 1, its density cannot be considered. Thus, we identify the entry with the smallest persistence value, denoted as $Et[h]$. To maximize the protection of already stored flows, we overwrite $BK[m].Et[h]$ with a probability of $1/BK[m].Et[h].p$. Note that a protected flow (i.e., $Flag[OF]$ is set to 1) cannot be replaced. Such an approach implies that a flow with a large persistence value will be hard to replace. In our preliminary experiments, we find that all flows evicted have a persistence below 4, with most at 1 or 2. It strengthens the protection of persistent flows in their early stages. *Contend* will bring "Ejection Errors", the main source of error for sketch-based solutions, which we analyze in Section 5.

4.2.4 Query. A query can be executed at any time. The Competition Layer will be traversed to find each position where $FP > 0$. If $Flag[OF] = 1$ in that entry, the flow is persistent. Then, we locate its ID in the Protection Layer to retrieve the overflow counts. Denote V_{PL} as the counter value in the Protection Layer and V_{CL} as the counter value in the Competition Layer. The final value V can be calculated as $V = V_{PL} \times 2^{L_c} + V_{CL}$, where L_c denotes the bit-length of the counter in the Competition Layer.

In this paper, we utilize an 8-bit counter for frequency (f) and a 6-bit counter for persistence (p) in the Competition Layer. Suppose their values are $f = 5$ and $p = 18$, while the corresponding Protection Layer values are $f_{of} = 1$ for frequency and $p_{of} = 3$ for persistence. The frequency of the flow, f_e, can be computed as: $f_e = 1 \times 2^8 + 5 = 261$. Similarly, the persistence, p_e, is calculated as: $p_e = 3 \times 2^6 + 18 = 210$. Thus, the density of the flow d_e is given by: $d_e = \frac{f_e}{p_e} = 1.243$. Any flow with a density below the threshold d_0 is reported as a PS flow, while other flows are considered persistent.

4.3 Optimization

4.3.1 Prune. In our observations for the majority of cases, if a flow is reported as a PS flow, it will not have a sudden increase in frequency throughout the entire timeline. Also, in our application scenario, a latent attack should not generate a high volume of packets at any time. Therefore, if an entry in PL contains an overflow counter $f_{of} = \mu$ larger than its $p_{of} = \tau$, the density will be $d_e > \frac{2^8\mu}{2^6(\tau+1)-1} = \frac{256\mu}{64(\tau+1)-1} > 4(\mu > \tau,\ \mu, \tau \in \mathbb{Z}^+)$. As a result, the flow can be cleared from both CL and PL even though it is being protected.

4.3.2 Burst Elimination. Interestingly, we have found that several flows generate a high volume of traffic in a short period but exhibit PS flow characteristics during all other time intervals. With their relatively high overall density, these flows may not meet the PS flow criterion for the entire period. With the aim of reporting these flows for security, if the overflow counter for f increases more than twice in one time window, we only add 2 to the counter f_{of}.

4.3.3 One-time Traversal. During *Insert*, we need to traverse $BK[m]$ up to three times in order to search for FP_e, an empty slot, and the flow with the smallest persistence. In practical use, we introduced three extra counters at the CL for each bucket (not each entry): an empty slot indicator $Ep[m]$, a replacement indicator $Rp[m]$, and a replacement counter $MinP[m]$, all initialized to -1, as shown in Figure 5. They will help us find all three locations in one loop.

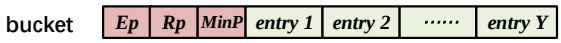


Figure 5: Three Extra Counters for Each Bucket.

4.4 A Running Example

Here, we present a simplified example of PSSketch operations. The green part represents one of the buckets in the Competition Layer with four entries, while the blue part represents the Protection Layer. The width of the counter f is 8 bits; thus, its overflow value is 256. In contrast, the overflow value of counter p is set to the persistence threshold of 50 in our example. The following describes how we handle incoming flows:

Example for Competition Layer. As shown in Figure 6, when e_1 comes for the first time and is hashed to this bucket, it searches for an empty entry and records $\langle FP_{e_1}, 1, 1, \boxed{10}\rangle$. Then flow e_1 arrives for the second time within this time window; its counter f is incremented by one while its counter p remains unchanged. Subsequently, a new flow e_3 hashes to the same bucket; however, due to the lack of empty entries, *contend* is initiated against e_1, which has the lowest value of counter p. Since $p_{e_1} = 1$, the replacement probability is $1/p_{e_1} = 1$, resulting in e_1 being replaced by e_3.

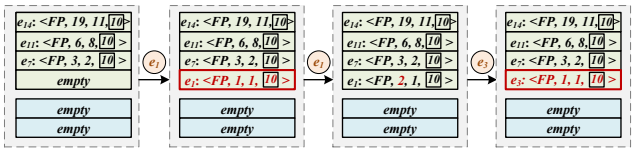


Figure 6: An Example of *Insert*, *Update* and *Contend*.

Example for Reporting to Protection Layer. After the last update of flow e_2, as shown in Figure 7, it reaches the overflow value of counter p. It is, therefore, reported to the Protection Layer with initial information $\langle ID_{e_2}, 0, 1\rangle$. On the other hand, flow e_5 comes and makes the counter f to be 256. Thus, it reports an overflow of counter f. Upon reporting, its information in the Protection Layer updates to $\langle FP_{e_5}, 4, 3\rangle$, satisfying the pruning condition $f_{of} > p_{of}$. As a result, its information is cleared from both layers.

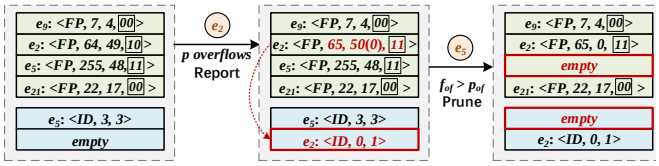


Figure 7: An Example of *Report* and *Prune*.

5 Mathematical Analysis

In this section, we first analyze the property of the density of a flow in 5.1. Then, we derive the error bounds for the PSSketch in 5.2. Finally, we analyze the time complexity and space complexity of PSSketch in 5.3. All the proof details can be found in Appendix B.

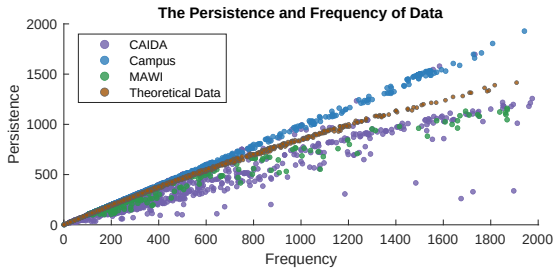


Figure 8: The Persistence and Frequency of Theoretical Data and the Data of CAIDA, Campus and MAWI.

In this section, we analyze the implementation of PSSketch and make the following two assumptions:

(1) The majority of flows in the data stream are neither persistent nor sparse (see Figure 3 for supporting evidence). Consequently, PS flows constitute only a small fraction of the data stream.
(2) We assume that all flows are mutually independent. For each flow, it is independent and identically distributed (i.i.d.) within each window, following a *Poisson* distribution with parameter λ, where the parameter λ follows a normal distribution. Figure 8 demonstrates that our assumptions align with the flow distribution in the three datasets we used (i.e., CAIDA, Campus, and MAWI).

5.1 Property of the Density of a Flow

Based on the given assumption, for a specific flow e (with the corresponding *Poisson* distribution parameter λ), let n_i represent the times that flow e appears in the i^{th} window.

Theorem 5.1. *The expectation and variance of frequency, persistence, and density of e after the i^{th} window is given by:*

$$\begin{aligned}\mathbb{E}[f_i] &= i\lambda \\ \mathbb{E}[p_i] &= i\cdot\left(1-e^{-\lambda}\right) \\ \mathbb{E}[d_i] &= \frac{\lambda}{\left(1-e^{-\lambda}\right)}.\end{aligned} \tag{1}$$

$$\begin{aligned}\mathbb{VAR}[f_i] &= i\lambda \\ \mathbb{VAR}[p_i] &= i\cdot e^{-\lambda}\cdot\left(1-e^{-\lambda}\right) \\ \mathbb{VAR}[d_i] &\le \frac{\lambda^2}{\left(1-e^{-\lambda}\right)^2}+\frac{\lambda+\lambda^2}{\left(1-e^{-\lambda}\right)} < \infty.\end{aligned} \tag{2}$$

According to L_p Convergence Theorem and Dominated Convergence Theorem, we can acknowledge that

$$\lim_{i\to\infty}\mathbb{E}[|d_i-\frac{\lambda}{\left(1-e^{-\lambda}\right)}|^2]=0, \tag{3}$$

which demonstrates that d_i is L_2 and almost surely converge to $\frac{\lambda}{(1-e^{-\lambda})}$.

Due to the constraint on p_0, the λ parameter of the Poisson distribution in the PS flows should neither be too small nor excessively large, balancing the need to identify low-density flows.

5.2 Error Bound Analysis

For PSSketch, the sources of error come from **Ejection Error**.

Theorem 5.2. *$\hat{d}_i$ is an unbiased estimator of $\frac{\lambda}{(1-e^{-\lambda})}$, and $\hat{d}_i$ L_2-convergence as well as almost surely convergence to $\frac{\lambda}{(1-e^{-\lambda})}$.*

Theorem 5.3. *$\hat{d}_i - d_i$ is an unbiased estimator of 0, and $\hat{d}_i - d_i$ L_2-convergence as well as almost surely convergence to 0.*

This demonstrates that, when the number of windows is sufficiently large, the error caused by ejection converges to zero. This means that even if a suspected PS flow is ejected, it will eventually be reported as a PS flow as the number of windows is large enough.

5.3 Complexity Analysis

5.3.1 Time Cost of PSSketch. In the Insert phase, a hash function is applied once for both the competition and protection layers, each with a time complexity of $O(1)$.

5.3.2 Space Cost of PSSketch.

Theorem 5.4. *The total storage space required by PSSketch is*

$$XY\cdot(L_c^{FP}+L_c^{f}+L_c^{p}+L_c^{flag})+R\cdot(L_c^{ID}+L_c^{f_{of}}+L_c^{p_{of}}), \tag{4}$$

where $R \ll X\cdot Y$.

Theorem 5.5. *The maximum frequency and the maximum persistence of a flow that can be stored in PSSketch is*

$$f_{max}=(2^{L_c^{f_{of}}}-1)\cdot(2^{L_c^{f}}-1)\approx 2^{L_c^{f_{of}}+L_c^{f}}, \tag{5}$$

$$p_{max}=(2^{L_c^{p_{of}}}-1)\cdot(2^{L_c^{p}}-1)\approx 2^{L_c^{p_{of}}+L_c^{p}}. \tag{6}$$

Theorem 5.6. *If PISketch stores flow data with the same order of magnitude for the maximum frequency and persistence as PSSketch, then the Weight Sketch component of PISketch must have at least:*

$$XY \cdot \left(L_c^{ID} + \log_2 L \cdot \left(L_c^{f_{of}} + L_c^{f}\right) + \left(L_c^{p_{of}} + L_c^{p}\right)\right) \tag{7}$$

Theorem 5.7. *If the Weight Sketch in PISketch and the Competition Layer in PSSketch occupy the same storage space, then:*

$$p_{max}^{PISketch} = 2^{L_c^N} - 1 \ll p_{max}. \tag{8}$$

The above arguments demonstrate that, compared to PISketch, PSSketch has a significant advantage in space complexity due to its handling of overflows and the use of a dual-layer mechanism consisting of the Competition Layer and Protection Layer.

6 Evaluation

6.1 Setup

We implement PSSketch in C++ with Bob Hash [1]. The experiments are run on a PC with AMD R9 7940H, 16 cores, and 32 GB DRAM. We have released our source code on the Website [5] and the GitHub [4]. The datasets we used are as follows:

CAIDA. It is a set of anonymous IPs collected in 2018 [2] that contains 2,490K packets with 109,534 flows. The proportion of PS flow is 1.055%.

Campus. This dataset is collected from a real network within a campus, utilizing the same data as [49]. It contains 10,000K packets, comprising 259,948 distinct flows, with PS flows accounting for 1.433% of the total flows.

MAWI. A real traffic trace provided by the MAWI Working Group [3]. It contains 2,000K packets with 200,471 flows. PS flows only take 0.132% total.

6.2 Comparing Solutions and Metrics

In this study, we will compare PSSketch with three distinct models:

Strawman Solution: As introduced in Section 2.2, we utilize CMSketch to estimate frequency and On-off Sketch to estimate persistence, while using an extra array to store information of PS flows.

PISketch: PISketch [15] represents the current SOTA in detecting PS flow specifically. PSSketch employs a traditional sketch data structure and defines the concept of "weight" as a criterion for PS flows.

PISketch-Density: This model is based on PISketch, with modifications made to its criterion. It now selects PS flows by our proposed definition, "density" instead of "weight", while all other processing procedures remain unchanged. By comparing this model, we aim to illustrate the structural optimizations introduced by PSSketch.

By comparing the PS flows reported by different solutions with the actual PS flows in the datasets, we will derive the following two metrics through comparison with the answer set:

F1 Score: The calculation of the F1 Score is given by the formula $F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$, where *Precision* means the proportion of PS flows in the predicted set and *Recall* shows the proportion of PS flows successfully predicted in the answer set.

ARE: The calculation of the Average Relative Error (ARE) is defined as $ARE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right|$, where $\hat{y}_i$ is the value stored in the sketch and y_i is its actual value. n shows the total flows stored at the end.

Throughput: We will also evaluate the throughput of the three models, defined as the number of packets processed per second.

6.3 Parameter Evaluation

Our model incorporates three variables: the persistence threshold p_0, the density threshold d_0, and the aspect ratio of the Competition Layer ($X : Y$). This experiment is mainly used to test the sensitivity and robustness of PSSketch in different scenarios and workloads. Considering the different loads of various datasets, under the CAIDA and Campus datasets, we set minimal memory to 50 KB, small memory to 100 KB, and large memory to 150 KB. Under MAWI, the three values correspond to 15 KB, 25 KB, and 50 KB, respectively.

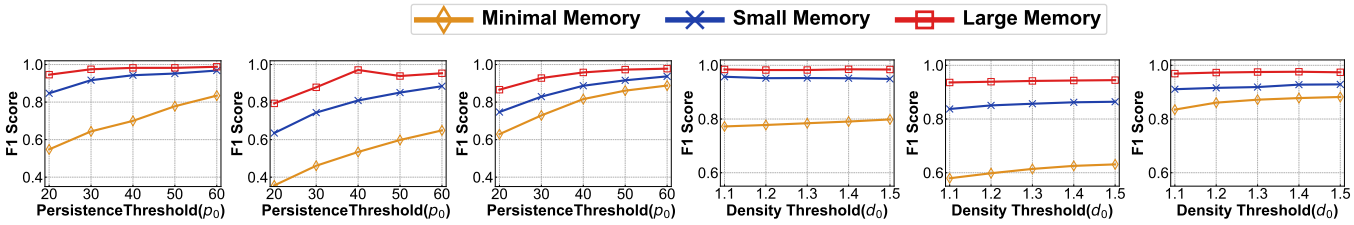
The impact of the persistent threshold p_0 and the density threshold d_0 on the F1 Score under varying memory constraints Mem is illustrated in Figure 9. The width Y is now set to 32, while $X = \frac{Mem}{32}$. As p_0 increases, the threshold for entering the Protection Layer rises, resulting in a significant enhancement of the F1 Score, particularly pronounced under memory-constrained conditions. This observation suggests that PSSketch performs better in more stringent threat detection scenarios. In contrast, the impact of the threshold d_0 on the F1 Score is relatively modest, yet it exhibits a positive correlation.

Figure 10(a), 10(b) and 10(c) illustrate the influence of p_0 on the average relative error. The optimal threshold p_0 varies according to load intensity; for higher intensities, such as in the Campus dataset, the optimal p_0 is lower. Conversely, as the load intensity decreases, the optimal threshold p_0 gradually increases.

Figure 10(d),10(e) and 10(f) depict the effect of p_0 on throughput. In the CAIDA and Campus datasets, stricter criteria significantly enhance the throughput of PSSketch, especially under conditions of ample memory. However, in the MAWI dataset, characterized by the lowest load intensity, the improvements in throughput are not obvious. Overall, the findings indicate that PSSketch achieves higher F1 Scores and throughput when addressing more rigorous threat detection tasks.

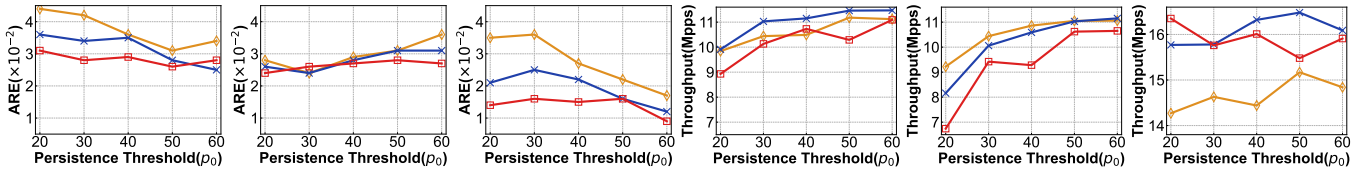
Note that in the experiment mentioned above, we do not present the impact of the threshold D on ARE and throughput. This is because D primarily influences *query*, not *insert*. Consequently, the effect of D on these two metrics is minimal.

We designate the number of entries Y within each bucket as the independent variable, thereby establishing $X = \frac{Mem}{Y}$. Figure 11 shows the impact of aspect ratio, where the X-axis represents y, the factor of Y, which satisfies $Y = 2^{y+1}$ for visual presentation. When we increase the bucket width Y while reducing the number of buckets $X = \frac{Mem}{Y}$, the F1 Score initially rises and then declines. Interestingly, under varying memory constraints, the optimal bucket width for all datasets consistently falls within the 16-32 range, with a notable concentration around 16. From the overall trend of ARE, it is evident that increasing the capacity of each bucket, rather than simply increasing the number of buckets, is more effective in reducing the average relative error. Moreover, increasing the number of entries per bucket leads to more memory access per iteration. As a result, it is reasonable that throughput decreases as the bucket width Y increases.



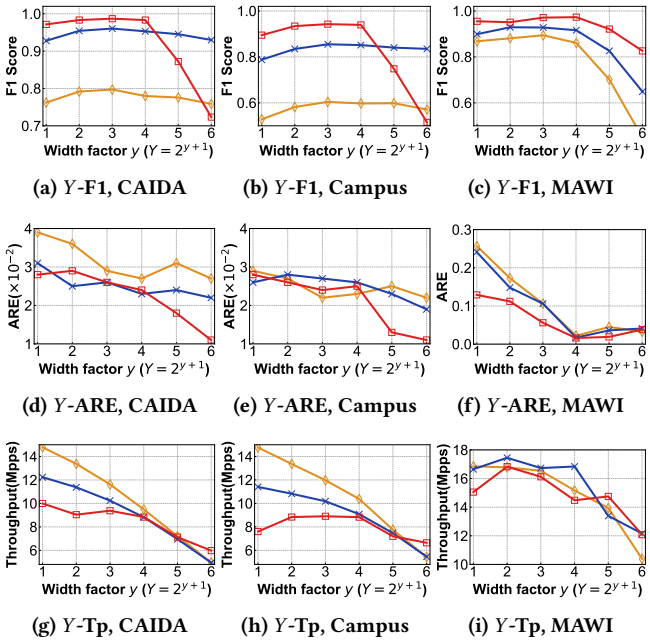
(a) p_0-F1, CAIDA (b) p_0-F1, Campus (c) p_0-F1, MAWI (d) d_0-F1, CAIDA (e) d_0-F1, Campus (f) d_0-F1, MAWI

Figure 9: The Impact of Persistence Threshold p_0 and Density Threshold d_0 on F1 Score.



(a) p_0-ARE, CAIDA (b) p_0-ARE, Campus (c) p_0-ARE, MAWI (d) p_0-Tp, CAIDA (e) p_0-Tp, Campus (f) p_0-Tp, MAWI

Figure 10: The Impact of Persistence Threshold p_0 on ARE and Throughput.



(a) Y-F1, CAIDA (b) Y-F1, Campus (c) Y-F1, MAWI

(d) Y-ARE, CAIDA (e) Y-ARE, Campus (f) Y-ARE, MAWI

(g) Y-Tp, CAIDA (h) Y-Tp, Campus (i) Y-Tp, MAWI

Figure 11: Impact of Aspect Ratio on F1, ARE, Throughput.

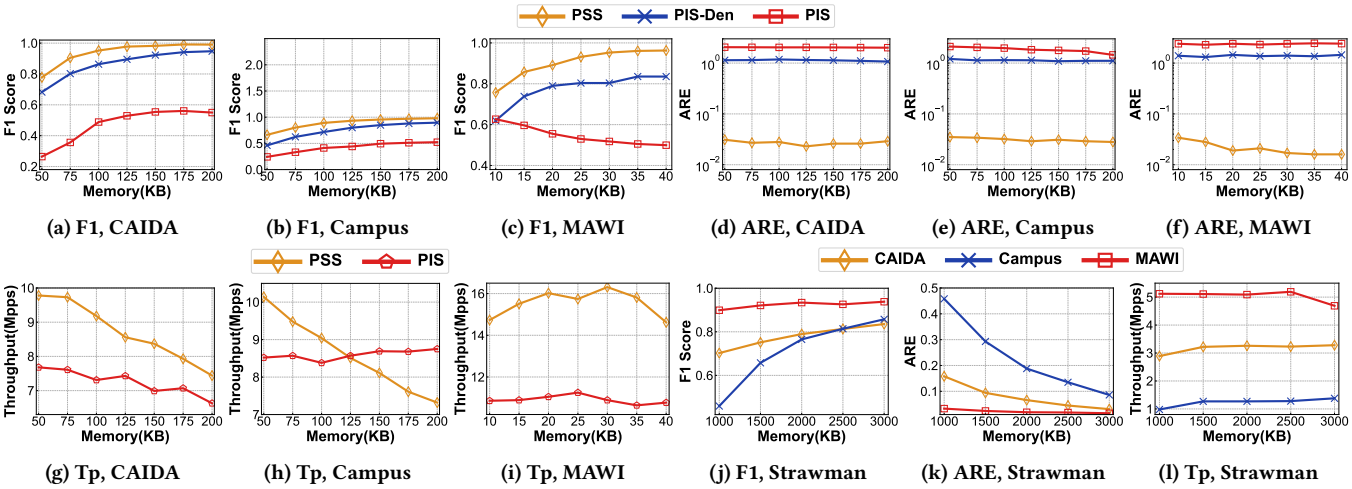
6.4 Performance

In this experiment, we selected $p_0 = 50$, $d_0 = 1.2$, and $Y = 32$ as the testing parameters and compared the various metrics of PSSketch against other solutions. It is important to emphasize that the optimal parameter combination may vary depending on the dataset and application scenario. We cannot tailor the parameter selection specifically for the dataset we used. Thus, the parameters chosen in this experiment represent a relatively optimal level under the current dataset, while the potential performance ceiling is higher than the results we present.

Figure 12(a), 12(b), 12(c) present a comparison of the F1 Scores under varying memory constraints. In all three datasets, PSSketch significantly improved the accuracy of PS flow finding, with F1 Score increases of 1.80x-2.94x, 1.87x-2.73x, and 1.21x-1.93x, compared to PISketch, respectively. After modifying PISketch's definition of PS flow, the F1 Score still rises to 1.44x. This demonstrates that not only does our model benefit from a more precise definition, but our data structure also exhibits clear advantages in filtering PS flows.

Figure 12(d), 12(e), 12(f) display the ARE under different memory conditions. Compared to PISketch, our method offers a 1-2 order of magnitude improvement. Specifically, under the higher load of the Campus dataset, the error decreases to 1.58%-1.93%, while in the lowest-load MAWI dataset, the error is reduced to 0.66%-1.42%. After adjusting the criterion for PISketch, we maintain at least a one-order-of-magnitude advantage in ARE, further supporting the high precision of PSSketch's statistical processing.

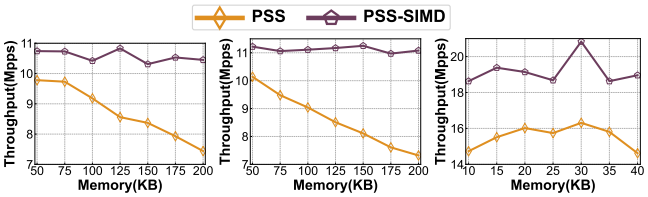
Additionally, Figure 12(g), 12(h), 12(i) illustrate the throughput comparison between PSSketch and PISketch. Since PISketch-Density only differs in the filtering process of PS flows and does not alter the data processing, its throughput is identical to that of PISketch. Experimental results demonstrate that PSSketch achieves a significant advantage under medium to low loads across all memory constraints. Specifically, throughput in the CAIDA dataset reaches 1.28x, and in the MAWI dataset, it reaches 1.50x. Under higher loads, PSSketch outperforms PISketch in scenarios with strict memory limitations, achieving a throughput of 1.19x while being slightly slower than PISketch when memory is more abundant. Across the three datasets, we observe a decreasing trend in throughput as memory increases, particularly in higher-load conditions. Section 6.6 proposes an optimization to mitigate this issue, aiming for superior performance across all datasets, thereby surpassing PISketch.



(a) F1, CAIDA (b) F1, Campus (c) F1, MAWI (d) ARE, CAIDA (e) ARE, Campus (f) ARE, MAWI

(g) Tp, CAIDA (h) Tp, Campus (i) Tp, MAWI (j) F1, Strawman (k) ARE, Strawman (l) Tp, Strawman

Figure 12: Comparison of Performance.



(a) Tp, CAIDA (b) Tp, Campus (c) Tp, MAWI

Figure 13: The Performance of PSSketch using SIMD.

Lastly, Figure 12(j), 12(k), and 12(l) present the performance of the Strawman solution. Since its valid memory range differs by an order of magnitude from PSSketch and PISketch, its performance is reported separately. Within the 1MB-3MB memory range, Strawman achieves F1 Scores of 0.70-0.84, 0.46-0.86, and 0.90-0.94 with the three datasets, notably lower than PSSketch. The corresponding ARE values are 1.11x-6.87x, 2.46x-16.36x, and 0.54x-2.06x compared to PSSketch. Although Strawman attains a higher F1 Score than PISketch and its ARE is generally on the same order of magnitude as that of PSSketch, it requires more than ten times the memory, and its throughput is only 13.39%–31.39% of that of PSSketch and 15.77%–45.19% of that of PISketch.

6.5 SIMD Optimization

To curb the trend of throughput decreasing with increasing memory, we utilize SIMD technology in the process of traversing the Competition Layer, which involves two main operations:

Window Flag Reset: At the beginning of each window, we need to reset the flag W of all entries in the M buckets to zero. By using SIMD, the reset for each bucket can be completed within 1-2 memory accesses, significantly reducing memory access time.

Bucket Search: For each flow e, the insertion process requires searching the corresponding bucket $BK[m]$. As discussed in Section 4.3, we previously optimized the three-loop process into a single loop. With the implementation of SIMD, this loop can be further reduced to 1-2 memory accesses, enhancing search efficiency.

As shown in Figure 13, the throughput of PSSketch on three datasets improves by up to 1.68x, 1.48x, and 1.85x, respectively, after integrating SIMD. More importantly, we eliminate the trend of decreasing throughput as memory size increases. That means, PSSketch demonstrates superior speed to the original PSSketch and strawman across all memory constraints and load intensities.

7 Conclusion

Finding persistent sparse flow is essential for network threat detection. In this paper, we first observe the distribution characteristics of different datasets. Then, we introduce a more accurate criterion for PS flows and propose PSSketch, a highly precise data structure. PSSketch employs a dual-layer structure with variable-length bit-level counters; it records frequency and persistence in the Competition Layer and counts overflow times in the Protection Layer, significantly enhancing memory efficiency. We use isolation protection and probability replacement to effectively protect potential PS flows, making it hard to replace them with a large number of regular flows. Additionally, optimizations such as pruning, burst elimination, and one-time traversal are applied to ensure high throughput. Experiments demonstrate that, compared to combined solutions and the SOTA method PISketch, we achieve up to 2.94x in F1 Score and a reduction in ARE by 1-2 orders of magnitude. Furthermore, our method has higher throughput across all scenarios. Overall, PSSketch provides a novel solution for finding persistent sparse flows with both high precision and efficiency.

Acknowledgments

Jiayao Wang finished this work during his internship at the Peng Cheng Laboratory, under the guidance of the corresponding authors Wenjun Li and Shuhui Chen. This work was supported in part by the Major Key Project of Peng Cheng Laboratory (PCL2023A06), the National Natural Science Foundation of China (U22A2036, 62202486, 62102203), the Key Research and Development Project of Jiangsu Province (BE2023004-4), the Science and Technology Innovation Program of Hunan Province (2024RC3139), the Young Top-notch Talent Project of Guangdong Province (2023TQ07X362), and the Basic Research Enhancement Program (2021-JCJQ-JJ-0483).

References

[1] 2025. BOBHash. http://burtleburtle.net/bob/hash/evahash.html.

[2] 2025. The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[3] 2025. MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/.

[4] Our GitHub. https://github.com/wenjunpaper/PSSketch.

[5] Our Website. https://wenjunli.com/PSSketch.

[6] Josep Aguilar-Saborit, Pedro Trancoso, Victor Muntes-Mulero, and Josep L. Larriba-Pey. 2006. Dynamic count filters. *Acm Sigmod Record* 35, 1 (2006), 26–32.

[7] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[8] Lu Cao, Qilong Shi, Yuxi Liu, Hanyue Zheng, Yao Xin, Wenjun Li, Tong Yang, Yangyang Wang, Yang Xu, Weizhe Zhang, and Mingwei Xu. 2024. Bubble Sketch: A High-performance and Memory-efficient Sketch for Finding Top-k Items in Data Streams. In *ACM CIKM*.

[9] Lu Cao, Qilong Shi, Weiqiang Xiao, Nianfu Wang, Wenjun Li, Zhijun Li, Weizhe Zhang, and Mingwei Xu. 2025. Hypersistent Sketch: Enhanced Persistence Estimation via Fast Item Separation. In *IEEE ICDE*.

[10] Saar Cohen and Yossi Matias. 2003. Spectral bloom filters. In *ACM SIGMOD*.

[11] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[12] James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourtchenko. 2019. Tuplemerge: Fast software packet processing for online packet classification. *IEEE/ACM Transactions on Networking* 27, 4 (2019), 1417–1431.

[13] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. Cuckoo Filter: Better Than Bloom. *login Usenix Magazine* 38, 4 (2013). https://www.usenix.org/publications/login/august-2013-volume-38-number-4/cuckoo-filter-better-bloom

[14] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking* 8, 3 (2000), 281–293.

[15] Zhuochen Fan, Zhoujing Hu, Yuhan Wu, Jiarui Guo, Sha Wang, Wenrui Liu, Tong Yang, Yaofeng Tu, and Steve Uhlig. 2023. PISketch: Finding Persistent and Infrequent Flows. *IEEE/ACM Transactions on Networking* 31, 6 (2023), 3191–3206.

[16] Michael T. Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. In *IEEE Allerton*.

[17] Thomas M. Graf and Daniel Lemire. 2020. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics* 25 (2020), 1–16.

[18] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. 2021. New directions in automated traffic analysis. In *ACM CCS*.

[19] Xiaoyang Hou, Jian Liu, Tianyu Tu, Rui Zhang, and Kui Ren. 2024. PrivRE: Regular Expression Matching for Encrypted Packet Inspection. In *IEEE ICDCS*.

[20] Jiawei Huang, Wenlu Zhang, Yijun Li, Lin Li, Zhaoyi Li, Jin Ye, and Jianxin Wang. 2022. ChainSketch: An efficient and accurate sketch for heavy flow detection. *IEEE/ACM Transactions on Networking* 31, 2 (2022), 738–753.

[21] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. 2003. Sketch-based change detection: Methods, evaluation, and applications. In *ACM IMC*.

[22] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. 2006. Data streaming algorithms for estimating entropy of network traffic. In *ACM SIGMETRICS*.

[23] Tao Li, Shigang Chen, and Yibei Ling. 2012. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking* 20, 5 (2012), 1622–1634.

[24] Wenjun Li, Dagang Li, Yongjie Bai, Wenxia Le, and Hui Li. 2019. Memory-efficient recursive scheme for multi-field packet classification. *IET Communications* 13, 9 (2019), 1319–1325.

[25] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. 2018. CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification. In *IEEE INFOCOM*.

[26] Weihe Li and Paul Patras. 2023. Tight-Sketch: A high-performance sketch for heavy item-oriented data stream mining with limited memory size. In *ACM CIKM*.

[27] Wenjun Li, Tong Yang, Ori Rottenstreich, Xianfeng Li, Gaogang Xie, Hui Li, Balajee Vamanan, Dagang Li, and Huiping Lin. 2020. Tuple Space Assisted Packet Classification With High Performance on Both Search and Update. *IEEE Journal on Selected Areas in Communications* 38, 7 (2020), 1555–1569.

[28] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*.

[29] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. 2008. Counter braids: A novel counter architecture for per-flow measurement. In *ACM SIGMETRICS*.

[30] Lailong Luo, Pengtao Fu, Shangsen Li, Deke Guo, Qianzhen Zhang, and Huaimin Wang. 2023. Ark filter: A general and space-efficient sketch for network flow analysis. *IEEE/ACM Transactions on Networking* 31, 6 (2023), 2825–2839.

[31] Ruijie Miao, Zheng Zhong, Jiarui Guo, Zikun Li, Tong Yang, and Bin Cui. 2022. BurstSketch: Finding Bursts in Data Streams. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2022), 11126–11140.

[32] Anirudh Ramachandran, Srinivasan Seetharaman, Nick Feamster, and Vijay Vazirani. 2008. Fast monitoring of traffic subpopulations. In *ACM SIGCOMM*.

[33] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented sketch: Faster and more accurate stream processing. In *ACM SIGMOD*.

[34] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. 2004. Reversible sketches for efficient and accurate change detection over network data streams. In *ACM IMC*.

[35] Qilong Shi, Chengjun Jia, Wenjun Li, Zaoxing Liu, Tong Yang, Jianan Ji, Gaogang Xie, Weizhe Zhang, and Minlan Yu. 2024. BitMatcher: Bit-level counter adjustment for sketches. In *IEEE ICDE*.

[36] Qilong Shi, Yuchen Xu, Jiuhua Qi, Wenjun Li, Tong Yang, Yang Xu, and Yi Wang. 2023. Cuckoo counter: Adaptive structure of counters for accurate frequency and top-k estimation. *IEEE/ACM Transactions on Networking* 31, 4 (2023), 1854–1869.

[37] Lu Tang, Qun Huang, and Patrick P.C. Lee. 2019. MV-Sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. In *IEEE INFOCOM*.

[38] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *USENIX NSDI*.

[39] Yao Xin, Chengjun Jia, Wenjun Li, Ori Rottenstreich, Yang Xu, Gaogang Xie, Zhihong Tian, and Jun Li. 2025. A Heterogeneous and Adaptive Architecture for Decision-Tree-Based ACL Engine on FPGA. *IEEE Trans. Comput.* 74, 1 (2025), 263–277.

[40] Yao Xin, Wenjun Li, Chengjun Jia, Xianfeng Li, Yang Xu, Bin Liu, Zhihong Tian, and Weizhe Zhang. 2024. Recursive Multi-Tree Construction With Efficient Rule Sifting for Packet Classification on FPGA. *IEEE/ACM Transactions on Networking* 32, 2 (2024), 1707–1722.

[41] Yao Xin, Wenjun Li, Guoming Tang, Tong Yang, Xiaohe Hu, and Yi Wang. 2022. FPGA-based updatable packet classification using TSS-combined bit-selecting tree. *IEEE/ACM Transactions on Networking* 30, 6 (2022), 2760–2775.

[42] Yao Xin, Wenjun Li, Gaogang Xie, Yang Xu, and Yi Wang. 2023. A parallel and updatable architecture for FPGA-based packet classification with large-scale rule sets. *IEEE Micro* 43, 2 (2023), 110–119.

[43] Hao Xu, Harry Chang, Wenjun Zhu, Yang Hong, Geoff Langdale, Kun Qiu, and Jin Zhao. 2023. Harry: A Scalable SIMD-based Multi-literal Pattern Matching Engine for Deep Packet Inspection. In *IEEE INFOCOM*.

[44] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. HeavyGuardian: Separate and guard hot items in data streams. In *ACM SIGKDD*.

[45] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM*.

[46] Tong Yang, Alex X Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. 2016. A shifting bloom filter framework for set queries. *Proceedings of the VLDB Endowment* 9, 5 (2016), 408–419.

[47] Tong Yang, Lingtong Liu, Yibo Yan, Muhammad Shahzad, Yulong Shen, Xiaoming Li, Bin Cui, and Gaogang Xie. 2017. SF-sketch: A fast, accurate, and memory efficient data structure to store frequencies of data items. In *IEEE ICDE*.

[48] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An accurate algorithm for finding Top-k elephant flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858.

[49] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. 2017. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1442–1453.

[50] Jin Ye, Lin Li, Wenlu Zhang, Guihao Chen, Yuanchao Shan, Yijun Li, Weihe Li, and Jiawei Huang. 2022. UA-Sketch: An accurate approach to detect heavy flow based on uninterrupted arrival. In *ACM ICPP*.

[51] Yinda Zhang, Jinyang Li, Yutian Lei, Tong Yang, Zhetao Li, Gong Zhang, and Bin Cui. 2020. On-off sketch: A fast and accurate sketch on persistence. *Proceedings of the VLDB Endowment* 14, 2 (2020), 128–140.

[52] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. 2021. DHS: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *ACM SIGKDD*.

[53] Jincheng Zhong, Shuhui Chen, and Biao Han. 2023. FPGA-CPU Architecture Accelerated Regular Expression Matching With Fast Preprocessing. *Comput. J.* 66, 12 (2023), 2928–2947.

[54] Jincheng Zhong, Ziling Wei, Shuang Zhao, and Shuhui Chen. 2022. TupleTree: A high-performance packet classification algorithm supporting fast rule-set updates. *IEEE/ACM Transactions on Networking* 31, 5 (2022), 2027–2041.

[55] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. 2018. Cold filter: A meta-framework for faster and more accurate stream processing. In *ACM SIGMOD*.

A Detail Version of Definitions

Flow & Frequency: Consistent with the main content.

Time Window: Given a data flow $S = \{e_1, e_2, e_3, \ldots, e_N\}$ and a window size t, the elements of S can be partitioned into multiple subsets, where $T_0 = \{e_1, e_2, \ldots, e_t\}$, $T_1 = \{e_{t+1}, e_{t+2}, \ldots, e_{2t}\}$, and so forth. Each subset T_k contains t elements. An element e_i belongs to the k^{th} window if $e_i \in T_k$, or we say e_i appears in the k^{th} window of the data flow. Mathematically, the window is defined as:

$$T_k = \{e_{kt+1}, e_{kt+2}, \ldots, e_{(k+1)t}\} \quad \text{for} \quad 0 \le k < \left\lceil \frac{n}{t} \right\rceil. \tag{9}$$

Persistence: Let e_i be an element in a data flow $T = \langle T_0, T_1, ... \rangle$ which is partitioned by time windows. There exists a subset $T_i = \langle T_{i0}, T_{i1}, ... \rangle$ such that $e_i \in T_{i0}$, $e_i \in T_{i1}$, ... while e_i does not appear in any time window in set $T \backslash T_i$. We denote $|T_i|$, the number of elements in T_i, as the persistence of e_i. That indicates the number of distinct time windows where e_i appears at least once.

Density: Given a continuous sequence of τ windows, if the element e_i exists in at least one window, its density is defined as:

$$D_i^\tau = \frac{f_i^\tau}{p_i^\tau}, \tag{10}$$

where p_i^τ denotes the persistence of element e_i over the τ windows, representing the number of windows in which e_i appears, with $p_i^\tau > 0$, and f_i^τ represents the frequency of element e_i in these τ windows, where $f_i^\tau \ge p_i^\tau$. It reflects the average access intensity of the element e_i within the data flow over the specified τ windows.

B Details of Mathematical Analysis

B.1 Property of the Density of a Flow

Theorem B.1. *The expectation and variance of frequency, persistence, and density of e after the i^{th} window is given by:*

$$\mathbb{E}[f_i] = i\lambda, \quad \mathbb{E}[p_i] = i\left(1 - e^{-\lambda}\right), \quad \mathbb{E}[d_i] = \frac{\lambda}{\left(1 - e^{-\lambda}\right)}. \tag{11}$$

$$\begin{aligned} &\mathbb{VAR}[f_i] = i\lambda, \quad \mathbb{VAR}[p_i] = i \cdot e^{-\lambda} \cdot \left(1 - e^{-\lambda}\right), \\ &\mathbb{VAR}[d_i] \le \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2} + \frac{\lambda + \lambda^2}{\left(1 - e^{-\lambda}\right)} < \infty. \end{aligned} \tag{12}$$

Proof.

$$\mathbb{P}[n_i = k] = \frac{\lambda^k}{k!} e^{-\lambda}. \tag{13}$$

$$\mathbb{P}[n \ge 1] = 1 - \mathbb{P}[n_i = 0] = 1 - e^{-\lambda}. \tag{14}$$

In other words, the probability that flow e appears in each window is $1 - e^{-\lambda}$. Given that there are i windows in total, the persistence p of the flow follows a binomial distribution: $p \sim \text{Binomial}(i, 1 - e^{-\lambda})$.

$$\mathbb{E}[f_i] = \mathbb{E}[\sum_{j=1}^{j=i} n_j] = \sum_{j=1}^{j=i} \mathbb{E}[n_j] = i\lambda. \tag{15}$$

$$\mathbb{E}[p_i] = \mathbb{E}[\sum_{j=1}^{j=i} I_{[n_j \ge 1]}] = \sum_{j=1}^{j=i} \mathbb{P}[n_j \ge 1] = i(1 - e^{-\lambda}). \tag{16}$$

$$\begin{aligned} \mathbb{E}[d_i] &= \mathbb{E}\left[\frac{\sum_{j=1}^{j=i} n_j}{\sum_{j=1}^{j=i} I_{[n_j \ge 1]}}\right] = \mathbb{E}\left[\mathbb{E}\left[\frac{\sum_{j=1}^{j=i} n_j}{\sum_{j=1}^{j=i} I_{[n_j \ge 1]}} | \sum_{j=1}^{j=i} I_{[n_j \ge 1]}\right]\right] \\ &= \mathbb{E}\left[\mathbb{E}\left[\frac{\sum_{j=1}^{j=i} n_j I_{[n_j \ge 1]}}{\sum_{j=1}^{j=i} I_{[n_j \ge 1]}} | \sum_{j=1}^{j=i} I_{[n_j \ge 1]}\right]\right] = \frac{\lambda}{1 - e^{-\lambda}}. \end{aligned} \tag{17}$$

$$\mathbb{VAR}[f_i] = \mathbb{VAR}[\sum_{j=1}^{j=i} n_j] = \sum_{j=1}^{j=i} \mathbb{VAR}[n_j] = i\lambda \tag{18}$$

$$\mathbb{VAR}[p_i] = \mathbb{VAR}[\sum_{j=1}^{j=i} I_{[n_j \ge 1]}] = \sum_{j=1}^{j=i} \mathbb{VAR} I_{[n_j \ge 1]} = i \cdot e^{-\lambda} \cdot \left(1 - e^{-\lambda}\right) \tag{19}$$

$$\begin{aligned} \mathbb{VAR}[d_i] &= \mathbb{E}[d_i^2] - \mathbb{E}^2[d_i] = \mathbb{E}\left[\left(\frac{\sum_{j=1}^{j=i} n_j}{\sum_{j=1}^{j=i} I_{[n_j \ge 1]}}\right)^2\right] - \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2} \\ &= \mathbb{E}\left[\mathbb{E}\left[\left(\frac{\sum_{j=1}^{j=i} n_j}{\sum_{j=1}^{j=i} I_{[n_j \ge 1]}}\right)^2 | \sum_{j=1}^{j=i} I_{[n_j \ge 1]}\right]\right] - \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2} \end{aligned} \tag{20}$$

$$\begin{aligned} &\mathbb{E}\left[\left(\frac{\sum_{j=1}^{j=i} n_j}{\sum_{j=1}^{j=i} I_{[n_j \ge 1]}}\right)^2 | \sum_{j=1}^{j=i} I_{[n_j \ge 1]}\right] \\ &= \left(\mathbb{E}[\sum_{j=1}^{j=i} n_j^2 I_{[n_j \ge 1]}] + 2\mathbb{E}[\sum_{j<k}] n_j n_k I_{[n_j \ge 1]} I_{[n_k \ge 1]}\right) / \left(\sum_{j=1}^{j=i} I_{[n_j \ge 1]}\right)^2 \\ &= \left(\frac{\lambda + \lambda^2}{1 - e^{-\lambda}} \sum_{j=1}^{j=i} I_{[n_j \ge 1]} + \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2} \left(I_{[n_j \ge 1]}\right) \cdot \left(I_{[n_j \ge 1]} - 1\right)\right) \\ &/ \left(\sum_{j=1}^{j=i} I_{[n_j \ge 1]}\right)^2 \\ &= \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2} + \left(\frac{\lambda + \lambda^2}{1 - e^{-\lambda}} - \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2}\right) / \left(\sum_{j=1}^{j=i} I_{[n_j \ge 1]}\right) \end{aligned} \tag{21}$$

$$\begin{aligned} \mathbb{VAR}[d_i] &= \mathbb{E}\left[\frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2} + \left(\frac{\lambda + \lambda^2}{1 - e^{-\lambda}} - \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2}\right) / \left(\sum_{j=1}^{j=i} I_{[n_j \ge 1]}\right)\right] \\ &= \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2} + \left(\frac{\lambda + \lambda^2}{1 - e^{-\lambda}} - \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2}\right) \mathbb{E}\left[\frac{1}{\sum_{j=1}^{j=i} I_{[n_j \ge 1]}}\right] \\ &\le \frac{\lambda^2}{\left(1 - e^{-\lambda}\right)^2} + \frac{\lambda + \lambda^2}{1 - e^{-\lambda}} < \infty \end{aligned} \tag{22}$$

□

According to L_p Convergence Theorem we can acknowledge that $\lim_{i\to\infty}\mathbb{E}[|d_i - \frac{\lambda}{(1-e^{-\lambda})}|^2] = 0$, which demonstrates that d_i is L_2 and almost surely converge to $\frac{\lambda}{(1-e^{-\lambda})}$.

B.2 Error Bound Analysis

B.2.1 Errors of Flow Ejection.

Theorem B.2. *$\hat{d}_i$ is an unbiased estimator of $\frac{\lambda}{(1-e^{-\lambda})}$, and $\hat{d}_i$ L_2-convergence as well as almost surely convergence to $\frac{\lambda}{(1-e^{-\lambda})}$.*

Proof. According to the structure of PSSketch $\hat{d}_i$ represents the density of flow e, counted from the last kicked window. Without loss of generality, we assume that the last time flow e was kicked occurred at the t-th window, denoted as event K_t. Due to the stationary increment property of the d_i process, we can infer that

$$\hat{d}_i|K_t = d_{i-t} \quad t = 0, 1, 2, \ldots, i-1. \tag{23}$$

$$\mathbb{E}[\hat{d}_i] = \mathbb{E}[\mathbb{E}[\hat{d}_i|K_t]] = \mathbb{E}[\mathbb{E}[d_{i-t}|K_t]] = \frac{\lambda}{1-e^{-\lambda}}. \tag{24}$$

$$\mathbb{VAR}[\hat{d}_i] = \mathbb{E}[\hat{d}_i{}^2] - \mathbb{E}^2[\hat{d}_i] = \mathbb{E}[\mathbb{E}[\hat{d}_i{}^2|K_t]] - \mathbb{E}^2[\hat{d}_i] < \infty \tag{25}$$

According to L_p Convergence Theorem and Dominated Convergence Theorem, we can acknowledge that

$$\lim_{i\to\infty}\mathbb{E}[|\hat{d}_i - \frac{\lambda}{\left(1-e^{-\lambda}\right)}|^2] = 0, \tag{26}$$

which shows that $\hat{d}_i$ is L_2 and almost surely converge to λ. □

Theorem B.3. *$\hat{d}_i - d_i$ is an unbiased estimator of 0, and $\hat{d}_i - d_i$ L_2-convergence as well as almost surely convergence to 0.*

Proof.

$$\mathbb{E}[\hat{d}_i - d_i] = \mathbb{E}[\mathbb{E}[\hat{d}_i|K_t] - d_i] = \mathbb{E}[\mathbb{E}[d_{i-t}] - d_i] = 0 \tag{27}$$

$$\mathbb{VAR}[\hat{d}_i - d_i] = \mathbb{E}[\left(\hat{d}_i - d_i\right)^2] = \mathbb{E}[\hat{d}_i{}^2] + \mathbb{E}[d_i^2] - 2\lambda < \infty \tag{28}$$

According to L_p Convergence Theorem, we can acknowledge that

$$\lim_{i\to\infty}\mathbb{E}[|\hat{d}_i - d_i|^2] = 0, \tag{29}$$

which shows that $\hat{d}_i - d_i$ is L_2 and almost surely converge to 0. □

B.3 Complexity Analysis

B.3.1 Time Cost of PSSketch. In the Insert phase, a hash function is applied once for both the competition and protection layers, each with a time complexity of $O(1)$.

B.3.2 Space Cost of PSSketch.

Theorem B.4. *The total storage space required by PSSketch is*

$$XY \cdot (L_c^{FP} + L_c^f + L_c^p + L_c^{flag}) + R \cdot (L_c^{ID} + L_c^{f_{of}} + L_c^{p_{of}}), \tag{30}$$

where $R \ll X \cdot Y$.

Proof. In the Competition Layer, there are X buckets, each containing Y cells. Each cell occupies a space of $L_c^{FP}+L_c^f+L_c^p+L_c^{flag}$. In the Protection Layer, there is a single bucket containing R cells, and each cell occupies $L_c^{ID}+L_c^{f_{of}}+L_c^{p_{of}}$. Therefore, the total storage space required is:

$$XY \cdot (L_c^{FP} + L_c^f + L_c^p + L_c^{flag}) + R \cdot (L_c^{ID} + L_c^{f_{of}} + L_c^{p_{of}}). \tag{31}$$

□

Theorem B.5. *he maximum frequency and the maximum persistence of a flow that can be stored in PSSketch is*

$$f_{max} = (2^{L_c^{f_{of}}} - 1) \cdot (2^{L_c^f} - 1) \approx 2^{L_c^{f_{of}}+L_c^f}, \tag{32}$$

$$p_{max} = (2^{L_c^{p_{of}}} - 1) \cdot (2^{L_c^p} - 1) \approx 2^{L_c^{p_{of}}+L_c^p}. \tag{33}$$

Proof. According to the structure of PSSketch, the maximum frequency that can be stored is the product of the maximum count that the frequency counters in the Competition Layer can hold and the overflow count in the Protection Layer, i.e.,

$$f_{max} = (2^{L_c^{f_{of}}} - 1) \cdot (2^{L_c^f} - 1) \approx 2^{L_c^{f_{of}}+L_c^f}. \tag{34}$$

Similarly, the maximum persistence that can be stored is

$$p_{max} = (2^{L_c^{p_{of}}} - 1) \cdot (2^{L_c^p} - 1) \approx 2^{L_c^{p_{of}}+L_c^p}. \tag{35}$$

□

Theorem B.6. *If PISketch stores flow data with the same order of magnitude for the maximum frequency and persistence as PSSketch, then the Weight Sketch component of PISketch must have at least:*

$$XY \cdot \left(L_c^{ID} + \log_2 L \cdot \left(L_c^{f_{of}} + L_c^f\right) + \left(L_c^{p_{of}} + L_c^p\right)\right) \tag{36}$$

Proof. For the PISketch, the weight W must satisfy:

$$2^{L_c^W} = L \cdot \left(L_c^{f_{of}} + L_c^f\right), \tag{37}$$

which implies that L_c^W must be the smallest integer greater than or equal to $\log_2 L \cdot \left(L_c^{f_{of}} + L_c^f\right)$. Similarly, to store $p_{max} \approx 2^{L_c^{p_{of}}+L_c^p}$, the parameter L_c^N in PISketch must be at least $L_c^{p_{of}} + L_c^p$. Thus, the memory required for the Weight Sketch in PISketch is no less than:

$$XY \cdot \left(L_c^{ID} + \log_2 L \cdot \left(L_c^{f_{of}} + L_c^f\right) + \left(L_c^{p_{of}} + L_c^p\right)\right). \tag{38}$$

□

Theorem B.7. *If the Weight Sketch in PISketch and the Competition Layer in PSSketch occupy storage space of the same order of magnitude, then:*

$$p_{max}^{PISketch} = 2^{L_c^N} - 1 \ll p_{max}. \tag{39}$$

Proof. In this case, we have:

$$p_{max}^{PISketch} = 2^{L_c^N} - 1 = 2^{L_c^p} - 1 \ll (2^{L_c^{p_{of}}} - 1) \cdot (2^{L_c^p} - 1) = p_{max}. \tag{40}$$

□

Overall, the above arguments demonstrate that,compared to PISketch, PSSketch has a significant advantage in terms of space complexity due to its overflows and the dual-layer mechanism.