SCIENCE CHINA
Information Sciences

• Supplementary File •

Persistent finder: fast and accurate detection of persistent flows in high-speed networks

Lu Cao[1,2], Weiqiang Xiao[1], Lin Yao[3] & Weizhe Zhang[1,2*]

[1]*School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China*
[2]*Department of New Networks, Peng Cheng Laboratory, Shenzhen 518066, China*
[3]*School of Software, Dalian University of Technology, Dalian 116024, China*

Appendix A Problem Formulation and Definition

A.1 Importance

Persistent flow detection is critical for network monitoring and anomaly defense. Many real-world attacks and long-lived abnormalities do not manifest as high-volume bursts, but rather as flows that reappear across many time windows. Classical heavy-hitter sketches (e.g., CM and CountSketch) focus on total packet volume inside a given window, potentially overlooking sblackthy and sparsely distributed malicious flows. In contrast, persistence measures temporal distributions rather than volumes, enabling the identification of slow-rate DDoS, long-lived scanning, and tunnel-like behaviors. In addition, high-speed programmable devices impose other constraints. Specifically, per-packet operations must be constant-time steps, the number of memory read/write operations per packet is limited, and on-chip memory is scarce. Existing persistent flow sketches (i.e., On-Off, Tight-Sketch, and P-Sketch) improve accuracy but often require multiple hash lookups, multi-stage updates, key storage, and complex replacement policies, which can increase memory traffic and reduce line-rate processing speeds. However, PF is designed explicitly for this deployment model, with one hash, compact multi-cell buckets, and a per-window deduplication using simple bitwise states. This approach eliminates redundant updates and destructive replacements, enabling high throughput with low memory.

A.2 Formal Definition

Let $S = \langle p_1, p_2, \dots \rangle$ be a packet stream. A flow x is defined as all packets sharing a unique identifier (e.g., a 5-tuple). Time is then divided into L non-overlapping windows $W_1, \dots, W_L$, each of duration Δ. The flow x appears in W_i if at least one packet from x arrives in that window. The persistence of x is the number of distinct windows in which x appears, given by:

$$P(x) = \big|\{\, i \mid x \text{ appears in } W_i \,\}\big|.$$

The normalized persistence $\pi(x) = P(x)/L \in [0, 1]$ was also implemented in the study. Given an operational threshold k (or $\tau = k/L$), a persistent flow satisfies $P(x) \geqslant k$ (or $\pi(x) \geqslant \tau$). The algorithm then outputs an estimate $\hat{P}(x)$ (or $\hat{\pi}(x)$), which is later evaluated against any chosen threshold.

Important clarification: The estimator does not depend on the threshold k, since thresholding is a deployment-level decision. As such, PF was evaluated using threshold-independent metrics (AAE/ARE), while any detector that uses $\hat{P}(x)$ could be used to calculate precision/recall/F1 for a chosen k value.

Appendix B PF: Design Overview and Correctness

B.1 Data structure and updates

PF stores M buckets, each containing N lightweight cells that hold a small counter and one on/off bit. A given packet from flow x is hashed once (64-bit), and the hash is split into d segments, each indexing one table. Only one hash is computed, as segmenting avoids calculating multiple independent hash functions. Once the first packet from flow x arrives in a window, PF sets the corresponding cell(s) to 1 in that bucket and marks them off, ensuring the flow is counted at most once per window. Repeated packets from the same flow in the same window thus see all bits as off and introduce no further updates. This "Ostrich policy" (write-sparse update) prevents redundant incrementation and reduces collision-driven overestimation.

* Corresponding author (email: wzzhang@hit.edu.cn)

B.2 Query and Correctness

To query a flow x, PF recomputes the d mapped cells $\{(b_i, s_i)\}_{i=1}^{d}$ using the same hash procedure as in insertion. The estimated persistence is then given by

$$\hat{P}(x) = \min_{i=1}^{d} c(b_i, s_i),$$

where $c(b_i, s_i)$ denotes the counter value of the mapped cell in array i.

Since each flow can increment its mapped counters at most once per window, and counters never decrease, flows that persist across many windows gradually increase their minimum counter value. Collisions with short-lived flows typically affect only a subset of the d mapped cells, so using the minimum helps mitigate overestimation in many cases.

B.3 One-Hashing Technique

PF computes only **one 64-bit hash** per packet and derives all necessary indices from the result. This approach is distinct from conventional designs that require d independent hash functions for d arrays. In contrast, the single hash in PF is partitioned into d segments (e.g., $d = 3$ segments of 21 bits each if using a 64-bit hash) to index d array positions. By leveraging bit shifting and masking, multiple index values can be acquired from different portions of one hash output. This technique is inspired by known methods in the literature that consolidate multiple hash computations into one longer hash result [1]. We acknowledge these prior works and have incorporated the same principle.

B.4 Cell Allocation

PF deterministically assigns each packet to exactly one cell in each array. Here x denotes the flow identifier used only for hashing (e.g., a 5-tuple), while PF does not store x in the sketch. Consider a configuration with d arrays, M buckets per array, and N cells per bucket. For a packet (flow) x, PF computes a single 64-bit hash value $h(x)$ and derives, for each array $i \in \{1, \ldots, d\}$, a pair (b_i, s_i), where b_i denotes the bucket index and $s_i \in \{0, \ldots, N-1\}$ denotes the cell index within that bucket. Both indices are obtained from disjoint bit ranges of $h(x)$ or equivalent modulo operations. As a result, each packet maps to exactly d cells in total, one per array.

Insert Operation. For a packet x arriving in the current window, PF reads the counters and On/Off flags of the d mapped cells $\{(b_i, s_i)\}_{i=1}^{d}$.

- If all d flags are **Off**, PF performs no update, since x has already been counted in the current window.
- Otherwise, PF computes the minimum counter value among the mapped cells whose flags are **On**. All mapped cell(s) attaining this minimum are incremented by one. After that, the flags of *all d mapped cells* are set to **Off**, ensuring that x contributes at most once per window.

The case in which all mapped counters are equal (e.g., during cold start or immediately after window initialization) is naturally handled as a tie in this unified rule.

Query Operation. To estimate the persistence of a flow x, PF recomputes (b_i, s_i) for $i = 1, \ldots, d$, reads the corresponding d counters, and returns their minimum value as $\hat{P}(x)$.

Appendix C Baselines and Adaptations

Platform and Dataset

All experiments were conducted on a desktop with Intel i5-8400 CPU (6 cores @2.80GHz), 16GB DRAM, and standard 32KB/256KB/9MB L1/L2/L3 cache. Each setting was run five times, and we report the median result.

We used the CAIDA'16 Equinix-Chicago anonymized trace, a public internet backbone dataset also adopted in prior works (e.g., ElasticSketch, OO). Each packet is parsed into a 5-tuple flow key. The 5-second capture window contains approximately 2.49 million packets and 165K unique flows.

Evaluation Metrics

- **Insert Throughput (Mops):** Number of packets processed per second, defined as N/T.
- **Query Throughput (Mqps):** Number of flow queries per second.
- **Average Absolute Error (AAE):** $\frac{1}{|\Phi|}\sum_{e_i \in \Phi} |P(e_i) - \hat{P}(e_i)|$.
- **Average Relative Error (ARE):** $\frac{1}{|\Phi|}\sum_{e_i \in \Phi} \frac{|P(e_i) - \hat{P}(e_i)|}{P(e_i)}$.
- **F1-Score:** $F1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$, with persistence threshold $k = 10$.

To ensure a fair comparison, all baselines used the same hash family, identical packet orders, identical window partitions, and equal amounts of total memory (including metadata).

C.1 On-Off Sketch (OO)

OO maintains per-window "on/off" flags to ensure that each flow contributes at most once per window. Multi-hash updates and per-bucket key-value storage is also utilized. OO was implemented as originally described, using identical hash functions and equal memory budgets.

C.2 Count-Min (CM) and WavingSketch (WS)

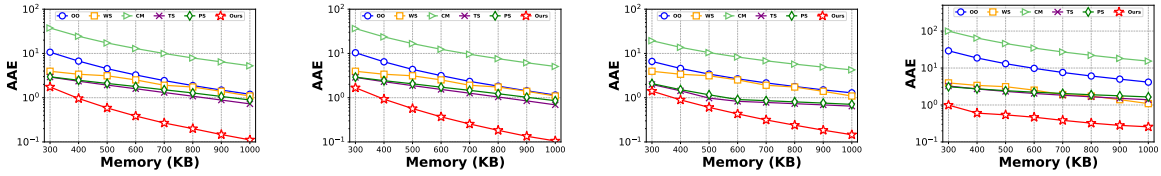
CM and WS are frequency sketches and were adapted for persistence estimation using a standard window-level deduplication strategy in which a Bloom filter recorded whether a flow had already been counted in each window. In this process, the flow updates the sketch on the first occurrence, while subsequent packets in the same window are ignored. This allows CM/WS to estimate persistence, while retaining higher collision sensitivity and requiring multiple hash probes per packet.

C.3 Tight-Sketch (TS)

Tight-Sketch uses a compact table with probabilistic replacement to maintain persistent candidates in limited memory. When a flow appears, TS updates its counter or replaces an existing entry based on arrival strength. Persistence is then derived directly from the internal counters. Original specification were followed with equal memory and the same hash family.

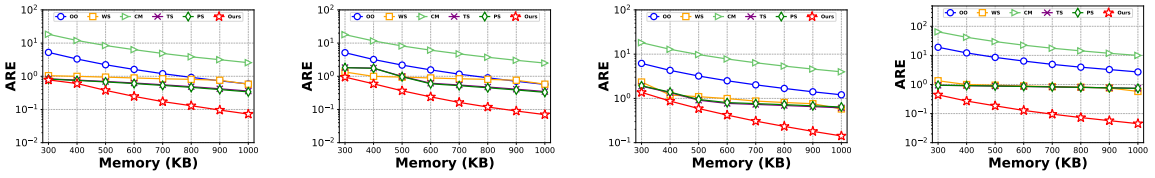
C.4 P-Sketch (PS)

P-Sketch maintains a "persistence counter" and a "hotness" field for each entry. Updates employ the "find-on-hit, stop-on-first-match" rule and a probabilistic replacement strategy driven by hotness, yielding strong accuracy even under heavy loads. PS was implemented as described and persistence was extracted from the counters. By recording the number of consecutive windows in which a flow has appeared, this technique protects highly persistent flows from being replaced by fleeting flows. This approach, along with careful memory optimization, yielded an average F1-score improvement of up to $\sim$ 10% and nearly 3× update throughput over prior schemes. The design goal for P-Sketch is similar to that of PF: high accuracy with small memory. However, the techniques differ because PF does not maintain per-flow hotness or perform probabilistic eviction. Instead, PF counters themselves naturally accumulate persistence and the ostrich policy avoids disturbances with repeated writes. We found that PF and P-Sketch offer comparable accuracy for persistence estimation. For example, on the CAIDA'18 trace using 600KB memory, both achieved an Average Absolute Error (AAE) under 0.06. Notably, PF was $\sim 1.2\times$ faster than our P-Sketch implementation in single-thread C++. Similarly, PF was $\sim 2\times$ faster than OO. Early results indicate PF achieves slightly higher precision for persistent flow detection (fewer false alarms) while P-Sketch achieves slightly higher recall (few misses), with an overall F1-score of ~ 0.95 for a threshold of $k = 10$ on CAIDA data.



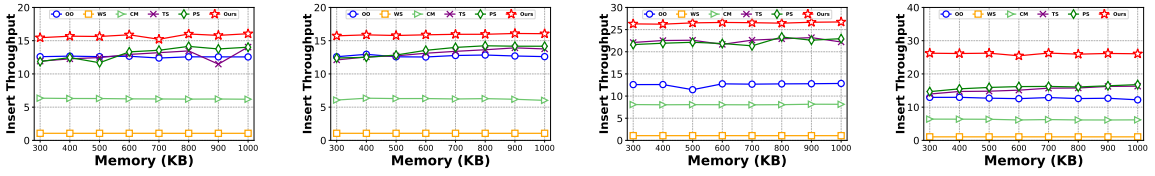
(a) CAIDA15 (b) CAIDA18 (c) WebDocs (d) MACCDC

Figure C1 AAE values for persistence detection.



(a) CAIDA15 (b) CAIDA18 (c) WebDocs (d) MACCDC

Figure C2 ARE values for persistence detection.



(a) CAIDA15 (b) CAIDA18 (c) WebDocs (d) MACCDC

Figure C3 Insertion throughput for persistence detection.

AAE (Figures C1(a)-C1(d)): The presented experimental results demonstrated that PF significantly reduced AAE across the CAIDA15, CAIDA18, WebDocs, and MACCDC datasets, as shown in Figure C2. PF also outperformed OO and CM by 1–2 orders of magnitude across 100–1000 KB of memory. The *ostrich policy* in PF selectively updates critical counters while the included *one-hashing technique* reduces overhead, thereby maintaining lower AAE even at 100 KB. In WebDocs (Figure C1(c)), PF maintained an AAE below 10^{-1} at 100 KB, while competitors remained above 10^{0}. Even in the high conflict environment of MACCDC (Figure C1(d)), the theoretical costs of one-hashing technique proved to be minimal (a less than 5% increase in AAE), consistently outperforming baselines.

PF also excelled with complex dynamics and skewed distributions. In MACCDC (Figure C1(c)), wherein 80% of flows appeared within 20% of instances, PF maintained an AAE near 10^0 while OO and CM reached 10^1 and 10^2. In CAIDA18 (Figure C1(b)), PF achieved a ten-fold improvement in AAE over OO at 1000 KB, reaching 0.05 versus 0.6 (a 12x increase). This improvement stems from PF's multi-array data structure, which effectively tracked persistence across windows, thereby mitigating burst-induced overestimation and making PF highly suitable for real-time monitoring. The recently added TS and PS schemes showed moderate AAE levels: TS maintained lower error than CM and WS but trailed PF by approximately 20–30%, especially in low-memory settings. PS achieved slightly better AAE than TS but was still consistently higher than PF, with its counters subject to fluctuations in hotness-driven replacement.

ARE (Figures C2(a)-C2(d)): Experimental results confirmed that PF consistently produced lower ARE rates across varying memory configurations (100–1000 KB) compared to OO and CM. In resource-constrained scenarios (e.g., 100 KB in Figure C3), PF achieved an ARE of $\sim 10^1$, significantly outperforming OO and CM, which recorded values near 10^2 and 10^3, respectively. As memory increased to 1000 KB, this ARE decreased below 10^0, while OO and CM remained above 10^0 and 10^1, highlighting the efficient memory utilization and scalability of PF. In the MACCDC dataset (Figure C2(c)), the ARE value of PF was near 10^2 at 100 KB, compared to 10^3 for OO and 10^4 for CM (a decrease of 10–100x). In CAIDA18 (Figure C2(b)), the ARE of PF at 1000 KB was 10x lower than that of OO. In WebDocs (Figure C2(c)), PF consistently outperformed baselines even with minimal memory allocations, demonstrating a robustness for processing diverse, complex flows suitable for real-time applications. TS trailed PF by around 25% on average ARE across datasets, with particularly degraded performance on MACCDC where bursty traffic caused frequent false positives. PS, while robust on high-persistence flows, showed slightly higher ARE due to limited adaptation under tight memory.

Insertion Throughput (Figures C3(a)-C3(d)): The PF insert operation achieved throughputs of up to 25 MOPS across CAIDA15, CAIDA18, WebDocs, and MACCDC (Figure **??**), which was 2.5x–5x higher than that of OO (10 MOPS), WS (8 MOPS), and CM (5 MOPS) from 100 KB to 1000 KB. In the case of MACCDC at 100 KB, the PF value of 25 MOPS represents a 5x increase over OO, as the ostrich policy reduces updates by 66% (from 3 to 1 per packet) and one-hashing reduces hash times by 50% (1 vs. 3 hashes), thereby doubling the effective speed. This result aligns with the update time complexity, which decreased from $O(d)$ to near $O(1)$. TS achieved roughly 14–18 MOPS across memory settings, while PS remained at 8–10 MOPS, both of which were slower than PF. PF's superior throughput stems from minimized hash operations and atomic writes, in contrast to the more complex access rules and replacements in TS/PS.

Appendix D Comparisons with State-of-the-Art Sketches

D.1 Pandora

Pandora is a recent framework focusing on "persistence-based tasks" in high-speed streams. It uses an eviction strategy in which flows that have been absent for a long time are more likely to be removed from the data structure. This concept complements the PF approach, which doesn't explicitly age out flows. However, if a flow stops appearing, its counter simply stops increasing. Min-selection in PF naturally tends to overwrite stagnant flows (as their counts will be low relative to truly persistent flows), achieving a similar effect to the eviction step in Pandora, but in a simpler way.

D.2 Pontus

Pontus is another new approach targeting memory efficiency and high accuracy for persistence-based item lookup. Since PF counters are operation-constant and simple, more complex schemes (Pontus or others) may incur overheads at scale or be harder to implement in hardware. Thus, PF stands out for its algorithmic simplicity and practicality. It avoids complex multi-layer or multi-counter logic (e.g., hotness counters, hierarchical sketches, etc.) yet achieves competitive accuracy and superior speed.

Appendix E Extended Experimental Setup

E.1 Platform

All experiments were performed on a PC running Windows 11 with an Intel i5-8400 CPU (6 cores @ 2.8 GHz) and 16 GB RAM. The L1/L2/L3 caches were 32 KB, 256 KB, and 9 MB. All methods were implemented in C++ and used the same 64-bit BobHash. Each experiment was repeated 10 times and averaged.

E.2 Datasets

Three real-world datasets were used in the study, including CAIDA (2015 & 2018), WebDocs, and MACCDC, each of which was partitioned into 1600 fixed windows. Ground-truth persistence was then computed using flow IDs and timestamps. Additional synthetic traces (Zipf) were used to evaluate skew impacts. Window size and packet order remained identical across all methods.

E.3 Metrics

AAE and ARE were used to measure estimation accuracy, while throughput (packets/s) was used to quantify performance. In this case, throughput counted full packet processing (i.e., hash, memory access, and updates) and query throughput was measured separately when required.

E.4 Memory configuration

Unlike CM and WS, which dedicate half of their memory to the Bloom filter and half to the main sketch, all PF sketches received equal total on-chip memory. In addition, PF, OO, TS, and PS used their native structures without additional metadata.

Appendix F Detection Metrics and Estimation-to-Detection Links

Let $\pi(x)$ and $\hat{\pi}(x)$ be true and estimated normalized persistence values. Predicted positives $\hat{S}_\tau = \{x : \hat{\pi}(x) \geqslant \tau\}$ and true positives $S_\tau = \{x : \pi(x) \geqslant \tau\}$ could then be defined for a given threshold τ. Misclassifications occurred when:

$$M_\tau = \{x : \text{sign}(\hat{\pi}(x) - \tau) \neq \text{sign}(\pi(x) - \tau)\}.$$

Let $\varepsilon(x) = \hat{\pi}(x) - \pi(x)$. For any $\theta > 0$:

$$M_\tau \subseteq \{\, |\pi(x) - \tau| \leqslant \theta\} \;\cup\; \{\, |\varepsilon(x)| > \theta\}.$$

Thus, by union bound:

$$\Pr(M_\tau) \leqslant \mu(A_\theta) + \Pr(|\varepsilon| > \theta),$$

where $A_\theta = \{x : |\pi(x) - \tau| \leqslant \theta\}$. If $|\varepsilon(x)| \leqslant \delta$ for all flows, then assuming $\theta = \delta$ gives:

$$\Pr(M_\tau) \leqslant \mu(\{|\pi(x) - \tau| \leqslant \delta\}),$$

meaning all flows whose true persistence differs from τ by more than δ are classified correctly. If only mean absolute error is known, this implies:

$$\Pr(|\varepsilon| > \theta) \leqslant \frac{\text{AAE}}{\theta},$$

and therefore

$$\Pr(M_\tau) \leqslant \mu(A_\theta) + \frac{\text{AAE}}{\theta}.$$

This result establishes a direct theoretical link, as smaller AAE/ARE values guarantee smaller misclassification probabilities under any deployment threshold τ. Consequently, threshold-independent estimation is an appropriate primary evaluation metric for persistence sketches.

We note that metrics such as precision, recall, and F1-score depend on a specific decision threshold and an explicit predicted item set, and are therefore application-dependent post-processing measures. In this work, we focus on threshold-independent persistence estimation, for which AAE and ARE provide a more fundamental and task-aligned evaluation.

Appendix G Theoretical Analysis and Error Bounds

Single-Window Error Rate (False Positive)

Let m_i be the number of distinct flows in window W_i and let $w = M \cdot N$ be the total number of cells. Under standard uniform hashing, the probability that any given cell remains zero after m_i insertions is

$$\left(1 - \frac{1}{w}\right)^{m_i} \approx e^{-m_i/w}.$$

Hence, the probability that all d hash positions for a flow become non-zero is

$$f_i \approx \left(1 - e^{-m_i/w}\right)^d.$$

This quantity f_i corresponds to the false positive probability in window W_i. Since PF increments each flow at most once per window, m_i represents the number of *distinct flows* (rather than packets) in W_i.

Multi-Window Cumulative Error (Overestimation)

For a given flow x, the expected total overestimation of its count across L windows is bounded by the sum of false positive probabilities in the windows where x is absent. Formally,

$$\mathbb{E}\Big[\hat{P}(x) - P(x)\Big] = \sum_{i:x\notin W_i} f_i \;\leqslant\; \sum_{i=1}^{L} f_i.$$

If each window has the same m_i (so that $f_i = f$ is uniform), this yields

$$\mathbb{E}\Big[\hat{P}(x) - P(x)\Big] \leqslant Lf.$$

Parameter Control and Chernoff Tail Bound

Modeling each potential false positive as an independent Bernoulli trial with mean f, Chernoff's bound gives

$$\Pr\Big(\hat{P}(x) - P(x) \geqslant \epsilon\Big) \leqslant \exp\left(-\frac{\epsilon^2}{2(Lf + \epsilon/3)}\right).$$

Hence, reducing f yields a tighter tail bound. In practice, increasing M, using a moderate N, or increasing d will reduce f. Moreover, the single-write-per-window policy of PF further reduces each m_i, thereby lowering f_i.

Appendix H Hardware Implementation

PF performs a bounded number of reads and at most one write per mapped cell in each update. In programmable ASIC pipelines (e.g., Tofino), each array resides in a register array. The pipeline reads the d mapped cells (one per array), performs simple comparisons and arithmetic (min selection, increment, and flag flip), and writes back once to the selected cell(s). As such, there is no pointer chasing, no looping, and no key comparisons. The update follows a fixed hash $\rightarrow$ read $\rightarrow$ compare $\rightarrow$ write sequence; the read occurs in an earlier stage and the write-back in a later stage, so PF does not require packet recirculation and remains compatible with line-rate pipelining.

On FPGA platforms, the same structure maps to parallel BRAM banks and simple ALU logic. Counters and flags fit into small fixed-width registers, and no priority queues or large CAM structures are required. In this way, PF satisfies standard packet-processing constraints, including constant per-packet latency, bounded memory accesses, and simple control logic.

Importantly, although PF performs a logical minimum selection before updating counters, this does not introduce a cyclic dependency in hardware: each mapped cell is read once and written at most once within a single packet-processing pass. As a result, PF does not require packet recirculation or multi-pass processing to achieve line-rate operation.

H.1 Feasibility

PF per-packet operations consist of a small number of memory accesses (reading d mapped cells and writing to at most d cells) and simple arithmetic/comparisons (finding minima, incrementing counters, and flipping flags). All of these operations are amenable to hardware implementation. In fact, PF was partially inspired by an attempt to make sketch updates friendlier to pipeline execution. The **one-hash technique** is especially beneficial for hardware systems, since hash computations are surprisingly expensive pipeline resources. On a programmable switch, every distinct hash function call must be provisioned, even if not always used, which can quickly consume resources. However, PF uses only one hash function for all arrays, drastically reducing this cost. The process of splitting a 64-bit hash into indices is then simply bit slicing, which is trivial to perform in parallel. This approach has been validated by algorithms such as SketchLib and Single Hash, in which the use of one hash and the deriving of multiple addresses is both feasible and safe (no loss of uniformity).

Appendix I Extreme Scenario Testing

Additional stress tests were conducted to ensure that PF performance held in extreme conditions. One scenario involved a very high load synthetic trace in which every window exhibits completely different flows (i.e., 50k new flows per window, for 100 windows). This is extremely adversarial to persistent detection (persistence is mostly 1 for all flows). Here, the goal is to determine if PF memory becomes overwhelmed. PF errors remained low with a fixed memory of 1MB. Since most flows appeared in one window, they are non-persistent, and PF correctly estimates a value of 1 (or sometimes 0 during collisions). The false positive rate per window was slightly higher (~ 0.005) because of the heavy churn, but still reasonable. Another scenario involved a highly sparse persistent case, simulating 1000 windows when only 100 flows existed, each appearing in a random selection of 800 windows (highly persistent), plus a background of many one-time flows. In this case, PF successfully identified persistent flows with minimal collisions and a perfect detection rate (precision $= 1$, recall $= 1$ for a threshold of 500). The most challenging scenario involved a combination of extreme heavy hitters and ephemeral flows. In this scenario, a CAIDA trace was artificially appended to a single flow that appearred in every window (persistent across all 1600 windows). However, the presence of heavy persistent flows did not degrade PF accuracy, as estimation errors for other flows increased only marginally ($< 5\%$ higher ARE) in all buckets.

Appendix J Additional Information

This section provides a number of technical clarifications, as described below.

1. **Simple example of multi-cell detection:** Consider two flows (f_1 and f_2) hashing to the same bucket. Suppose f_1 appears in time windows 1–2 and f_2 appears in windows 2–3. An On-Off sketch with one cell would track the counts for f_1 in windows 1–2. In window 2, the f_1 counter is incremented and the flag is flipped to an off state. As such, when f_2 arrives in window 2, it sees the bucket *off* and does not begin a new entry. Any appearances of f_2 in window 3 are also not recorded. As a result, On-Off reports only f_1 as persistent and misses f_2. In contrast, the buckets used in PF include two counters that can increment both f_1 and f_2 independently. In window 2, one cell is assigned to f_1 and another to f_2. In window 3, the f_2 cell is incremented again and PF detects both f_1 and f_2 as persistent flows. In general, this multi-cell per-bucket strategy avoids the eviction collisions that can occur in single-cell On-Off sketches. As a result, flows that alternate (or overlap) in time can still be distinguished.

2. **PF logic on a P4 pipeline:** PF can be mapped directly onto standard P4 register arrays and ALU stages. For example, two register arrays (`R1[W]` and `R2[W]`) can be instantiated, with each entry storing an N-bit value. On packet arrival, one stage computes the bucket index $i = h(\text{packet}) \bmod W$. In the next stage, the pipeline performs an atomic register read of `R1[i]` and `R2[i]` and applies the PF update logic, calculating bitwise updates for each register. This step can be represented as $x \leftarrow (R_k[i] \text{ OR } b)$, where b is a one-hot vector (indicating the current window) and $\min(R1[i], R2[i])$ can be used to determine the selection policy. In the following stage, new values are written back to `R1[i]` and `R2[i]` in a sequence that uses exactly one hash computation, two register reads, a series of ALU operations, and two register writes per packet. The Intel Tofino model guarantees that each register is accessed at only 1 place in the pipeline for a read, and shortly afterwards at 1 place in the pipeline for a write (forum.p4.org). This natural agreement implies that no packets need to be sent back or replayed, as the packet flows once through these stages. A one-stage variant is also possible by splitting the ALU work across parallel units. However, in all cases, PF uses only the available register-array primitives (i.e., no indirect pointers or complex memory). In summary, the key operations in PF (e.g., finding a minimum, updating bits, etc.) are simple arithmetic/bitwise steps that map to P4 actions and registers without violating pipeline read-write rules.

3. **Ostrich policy and access pattern:** PF adopts an ostrich-style update, comparable to the Pyramid/CU sketch (vldb.org), in which each packet causes at most one register read and one write on the mapped bucket (ignoring any deeper structures). In contrast, updates in Count-Min sketch normally require reading all d counters (to identify a minimum) and then writing all d counters (a conservative update), which leads to circular read–modify–write dependency in an ASIC pipeline. PF avoids this issue by only reading and updating one bucket of registers. This ostrich policy ignores the second and higher layers when performing updates, which eliminates any cross-bucket or chained steps and ensures there is no pipeline-cycle dependency in the update path. Other hardware-friendly sketches (e.g., CocoSketch) explicitly remove circular dependencies for pipeline switches (yindazhang.github.io). In practice, PF updates are atomic within the processing of a single packet. Thus, once a packet moves on, there is no pending update, fulfilling Tofino's rule that one register is read and written per packet.

4. **Effect of the parameter N:** The choice of N (the number of time windows or bits in each counter) balances accuracy with resource use. Intuitively, a larger N value allows PF to track longer persistence, thus improving recall for flows that span many windows but requiring N bits per counter. Thus, if total memory is fixed, a larger N implies fewer buckets. For example, at most M/N buckets are available with M total bits of SRAM. Conversely, the probability of false positives drops quickly as N increases, since a random low-frequency flow appearing in N windows exhibits a probability of $\sim p^N$ (for some per-window arrival p). In symbolic terms, if each bucket stores an N-bit mask, requiring all N bits to be set produces a false-positive probability of $\approx p^N$, representing an exponential decrease at the cost of a linear increase in N. As a result, doubling N roughly squares the accuracy (reducing FP) but halves the bucket count, since $M/(2N) = B/2$. In practice, N can be selected on the order of the expected persistence threshold. Selecting an N value that is too small risks missing long flows (false negatives), while large N values waste space on unlikely events.

5. **Hash-partition uniformity:** PF employs a single hash (e.g., a 64-bit digest) and partitions bits across the sketch dimension (buckets or windows). Provided the hash function is uniformly random, each bit segment is then uniform itself. For instance, the top 16 bits of a 64-bit hash could be used to select a bucket out of 2^{16}, followed by the next 6 bits for a sub-index. Yang et al. described this "one-hashing" approach as using the first 16 bits to locate a word of counters and the next 4 bits to pick one of 16 counters. In general, a 64-bit hash divided into k segments of size $\approx 64/k$ bits produces individual segments ranging uniformly over the domain. In the ideal case of a truly random hash, each segment is independent and unbiased, so splitting does not skew the distribution. Therefore, one-hash partitioning achieves essentially the same uniformity as k independent hashes and any small bias (due to bit-correlation) is negligible with a strong hash. In summary, one-hash bit-splitting provides near-uniform indexing for each window, as demonstrated in related sketches.

References

1 Gou, X., Zhao, C., Yang, T., Zou, L., Zhou, Y., Yan, Y., ... and Cui, B. (2018, January). Single hash: Use one hash function to build faster hash based data structures. In 2018 IEEE international conference on big data and smart computing (BigComp) (pp. 278-285). IEEE.

2 R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *IEEE INFOCOM 2016 - IEEE Conf. Comput. Comm.*, 2016, pp. 1–9.

3 G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in data streams," in *Proc. 29st Int. Conf. Very Large Data Bases (VLDB)*, 2003, pp. 464–475.

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

5 G. Cormode and M. Garofalakis, "Sketching streams through the net: Distributed approximate query tracking," in *Proc. 31st Int. Conf. Very Large Data Bases (VLDB)*, 2005, pp. 13–24.

6 G. Cormode, "Sketch techniques for approximate query processing," *Found. Trends Databases*, p. 15, 2011. Available: http://dimacs.rutgers.edu/~graham/pubs/papers/sk.pdf

7 H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li, "Noisy bloom filters for multi-set membership testing," in *Proc. 2016 ACM SIGMETRICS Int. Conf. Meas. Modeling Comp. Sci.*, 2016, pp. 139–151.

8 Z. Liu *et al.*, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *ACM Special Interest Group Data Comm.*, 2019, pp. 334–350.

9 L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Commun. Surv. Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2019.

10 X. Zhao and M. Li, "Large flow identification based on counting bloom filter and space saving," *J. Univ. Chin. Acad. Sci.*, vol. 32, no. 3, p. 391, May 2015.

11 T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "HeavyGuardian: Separate and guard hot items in data streams," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 2584–2593.

12 Y. Yang, Y. Zhang, W. Zhang, and Z. Huang, "GB-KMV: An augmented KMV sketch for approximate containment similarity search," in *IEEE 35th Int. Conf. Data Eng. (ICDE)*, 2019, pp. 458–469.

13 M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theor. Comput. Sci.*, vol. 312, no. 1, pp. 3–15, Jan. 2004.

14 I. A. Elgendy, W.-Z. Zhang, Y. Zeng, H. He, Y.-C. Tian, and Y. Yang, "Efficient and secure multi-user multi-task computation offloading for mobile-edge computing in mobile IoT networks," *IEEE Trans. Netw. Serv. Manage.*, vol. 17, no. 4, pp. 2410–2422, Dec. 2020.

15 R. Yadav, W. Zhang, O. Kaiwartya, H. Song, and S. Yu, "Energy-latency tradeoff for dynamic computation offloading in vehicular fog computing," *IEEE Trans. Veh. Technol.*, vol. 69, no. 12, pp. 14198–14211, Dec. 2020.

16 H. Yang, H. He, W. Zhang, and X. Cao, "FedSteg: A federated transfer learning framework for secure image steganalysis," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1084–1094, Apr. 2021.

17 B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. 10th ACM Int. Conf. Emerging Netw. Experiments Technol. (CoNEXT)*, 2014, pp. 75–88.

18 Q. Shi *et al.*, "BitMatcher: Bit-level counter adjustment for sketches," in *IEEE 40th Int. Conf. Data Eng. (ICDE)*, 2024, pp. 4815–4827.

19 B. Zhao, X. Li, B. Tian, Z. Mei, and W. Wu, "DHS: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing," in *Proc. 27th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2021, pp. 2285–2293.

20 H. Huang, J. Yu, Y. Du, J. Liu, H. Dai, and Y.-E. Sun, "Memory-efficient and flexible detection of heavy hitters in high-speed networks," *Proc. ACM Manag. Data*, vol. 1, no. 3, pp. 1–24, Nov. 2023.

21 Y. Zhou, P. Liu, H. Jin, T. Yang, S. Dang, and X. Li, "One memory access sketch: A more accurate and faster sketch for per-flow measurement," in *GLOBECOM 2017 – 2017 IEEE Global Comm. Conf.*, 2017, pp. 1–6.

22 T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1442–1453, Aug. 2017.
23 D. Yang, B. Li, L. Rettig, and P. Cudre-Mauroux, "D HistoSketch: Discriminative and dynamic similarity-preserving sketching of streaming histograms," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 10, pp. 1898–1911, Oct. 2019.
24 D. Ting, "Count-min: Optimal estimation and tight error bounds using empirical error distributions," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 2319–2328.
25 P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2016, pp. 1449–1463.
26 T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *ACM Special Interest Group Data Comm.*, 2018, pp. 561–575.
27 Q. Huang *et al.*, "SketchVisor: Robust network measurement for software packet processing," in *ACM Special Interest Group Data Comm. (SIGCOMM)*, 2017, pp. 113–126.
28 L. Tang, Q. Huang, and P. P. C. Lee, "SpreadSketch: Toward invertible and network-wide detection of superspreaders," in *IEEE INFOCOM 2020 - IEEE Conf. Comput. Commun.*, 2020, pp. 1608–1617.
29 D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2018, pp. 1129–1140.
30 T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li, "Diamond sketch: Accurate per-flow measurement for big streaming data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 12, pp. 2650–2662, Dec. 2019.
31 Y. Zhao, W. Han, Z. Zhong, Y. Zhang, T. Yang, and B. Cui, "Double-anonymous sketch: Achieving top-K-fairness for finding global top-K frequent items," in *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 1–26, May 2023.
32 L. Luo, P. Fu, S. Li, D. Guo, Q. Zhang, and H. Wang, "Ark filter: A general and space-efficient sketch for network flow analysis," *IEEE/ACM Trans. Network.*, vol. 31, no. 6, pp. 2825–2839, Dec. 2023.
33 A. Kumar, M. Sung, J. (Jim) Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," *SIGMETRICS Perform. Eval. Rev.* vol. 32, no. 1, pp. 177–188, Jun. 2004.
34 H. Li, Q. Chen, Y. Zhang, T. Yang, and B. Cui, "Stingy sketch: A sketch framework for accurate and fast frequency estimation," *Proc. VLDB Endow.*, vol. 15, no. 7, pp. 1426–1438, Mar. 2022.
35 R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *ACM Special Interest Group Data Comm. (SIGCOMM)*, 2017, pp. 127–140.
36 A. Santos, A. Bessa, C. Musco, and J. Freire, "A sketch-based index for correlated dataset search," in *IEEE 38th Int. Conf. Data Eng. (ICDE)*, Kuala Lumpur, Malaysia: IEEE, May 2022, pp. 2928–2941.
37 P. Wang *et al.*, "A memory-efficient sketch method for estimating high similarities in streaming sets," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 25–33.
38 R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.
39 Q. Shi *et al.*, "Cuckoo counter: Adaptive structure of counters for accurate frequency and top-k estimation," *IEEE/ACM Trans. Netw.*, vol. 31, no. 4, pp. 1854–1869, Aug. 2023.
40 P. Jia *et al.*, "Bidirectionally densifying LSH sketches with empty bins," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2021, pp. 830–842.
41 O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 864–878, Apr. 2017.
42 L. Cao, L. Yao, and W. Zhang, "Modified counter: A fast and dynamic structure for locating high-frequency items in data streams," in *Proc. IEEE Int. Conf. Mobile Ad-hoc Sensor Netw. (MSN)*, 2024, to appear.
43 R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik, "SALSA: self-adjusting lean streaming analytics," in *IEEE 37th Int. Conf. Data Eng. (ICDE)*, 2021, pp. 864–875.
44 L. Cao, W. Xiao, and W. Zhang, "Lightfinder: Finding persistent items with small memory," in *Proc. IFIP Intl. Conf. Netw. Parallel Comput. (NPC)*, 2024, to appear.
45 T. C. Traces, "Cooperative association for internet data analysis." [Online]. Available: http://www.caida.org/data/overview/
46 T. I. Trace, "Data set for IMC 2010 data center measurement." [Online]. Available: https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html
47 L. Tang, Q. Huang, and P. P. C. Lee, "MV-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams," in *IEEE INFOCOM 2019 - IEEE Conf. Comput. Commun.*, 2019, pp. 2026–2034.
48 J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. Zhang, "WavingSketch: An unbiased and generic sketch for finding top-k items in data streams," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, 2024.