PALLAS: A Data-Plane-Only Approach to Accurate Persistent Flow Detection on Programmable Switches in High-Speed Networks

Weihe Li[§*], Beyza Bütün[†#*], Tianyue Chu[‡], Marco Fiore[†], Paul Patras[§]
[§]The University of Edinburgh, [†]IMDEA Networks Institute, [#]Universidad Carlos III de Madrid, [‡]Telefónica Research
weihe.li@ed.ac.uk, beyza.butun@imdea.org, tianyue.chu@telefonica.com, marco.fiore@imdea.org, paul.patras@ed.ac.uk

***Abstract*—In high-speed data center networks, persistent flows are repeatedly observed over extended periods, potentially signaling threats such as stealthy DDoS or botnet attacks. Monitoring every flow in production-grade hardware switches that feature limited memory, however, is challenging under typical high flow rates and data volumes. To tackle this, approximate data structures, like sketches, are often employed. Yet many existing methods rely on per-time-window flag resets, which require frequent control-plane interventions that make them unsuitable for high-speed traffic. This paper introduces PALLAS, a fully data-plane-implementable sketch for detecting persistent flows in high-speed networks with high accuracy, obviating the need for time-window-based resets. We further propose OPT-PALLAS, an enhanced variant of PALLAS that improves detection accuracy by incorporating flow arrival patterns. We present a rigorous error bound analysis for both PALLAS and OPT-PALLAS, along with extensive performance evaluations using a P4-based prototype on an Intel Tofino switch. PALLAS scales persistent flow detection to line-rate capacity, while state-of-the-art solutions fail to operate beyond a few Mbps. Our results show that PALLAS and OPT-PALLAS can accurately detect persistent flows in traffic volumes over 60× higher than those handled by the best existing approach. Additionally, even under low-speed traffic, PALLAS and OPT-PALLAS achieve 4.21% and 7.85% higher lookup accuracy while consuming only 8.5% and 9.7% of switch resources, respectively. Extensive trace-driven results on a CPU platform further validate the high detection accuracy of OPT-PALLAS compared to existing methods.**

***Index Terms*—Persistent flows, programmable switches, high-speed networks, sketch, approximate data structure**

I. INTRODUCTION

Detecting persistent flows–those that remain active over extended periods–is vital for network management, recommendation systems, and security applications, such as identifying and mitigating stealthy data exfiltration and Advanced Persistent Threat (APT) activities [1]. For example, some adversaries send malicious traffic at a controlled, low rate over an extended duration to evade traditional volume-based anomaly detectors [2]. However, in high-speed networks, the sheer flow rates and massive data volumes make it impractical to track every flow within the constrained memory of switching infrastructure, even in cutting-edge programmable switches [3].

To overcome this challenge, a variety of sketch-based methods have been proposed [4]–[7]. A sketch is a hashing-based data structure that stores flow information in limited memory while retaining acceptable accuracy [8], [9]. Meanwhile, programmable switches are increasingly adopted in high-speed networks due to their rapid processing capabilities and high flexibility, enabling adaptable forwarding strategies through P4-based prototyping [10]. However, programmable switches still face stringent constraints—such as a limited number of pipeline stages and a narrow range of arithmetic operations [11]—which complicate the implementation of existing sketch-based methods for persistent flow detection.

Limitations of Existing Methods: (i) Existing sketch-based methods for persistent flow lookup use a flag to remove duplicates [4]–[6], [12]. When these methods are implemented on a programmable switch, a control plane logic needs to reset the flags periodically, which introduces a delay of typically tens of milliseconds [13]. At high flow rates this delay causes untimely flag resets, resulting in inaccurate persistence tracking and rendering existing methods ineffective. (ii) In a programmable switch, each packet traverses a pipeline with a fixed number of stages in a strictly unidirectional workflow—data flows from the first stage to the last [11] (e.g., Tofino 1 has 12 stages [14]). Because some sketching methods involve multiple circular dependencies, using multiple rows to store tracked items would exceed these stage constraints, prompting most existing designs to adopt a single-row approach [11], [12], [15]. However, a single-row sketch inevitably suffers from severe hash collisions when memory resources are constrained, causing persistent flows to be erroneously evicted by non-persistent ones—especially under common, highly skewed traffic distributions [5].

Our Solution: To surmount these constraints, we present PALLAS, a new sketch specifically designed for accurate persistent flow detection on programmable switches. *To the best of our knowledge, this is the first method for persistent flow detection that maintains reliable performance at high data rates in a production-grade switch.* We address current limitations as follows. (i) First, instead of relying on per-time-window flag resets—which necessitate communication between the control plane and the data plane—we eliminate the use of flags altogether. We track persistence by using a global counter to record the current time window and storing each flow's latest arrival time window in the sketch buckets. When a flow arrives, its tracked time window is compared against the global one to determine whether it is the flow's first arrival within that window. (ii) Second, under highly

* These authors contributed equally to this work.

979-8-3315-0376-5/25/$31.00 ©2025 IEEE

skewed traffic distributions, persistent flows are prone to being mistakenly evicted from buckets by a large number of non-persistent flows. To address this, we adopt a probability-based update mechanism that assigns lower eviction probabilities to flows with higher persistence, ensuring robust performance in practical, skewed traffic scenarios. We further introduce an optimized version of PALLAS, called OPT-PALLAS, which improves detection accuracy by incorporating the arrival patterns of persistent flows. In practice, truly persistent flows rarely remain inactive for extended periods [16]. Thus, when a flow is absent for a long duration, OPT-PALLAS increases the likelihood of evicting it from its bucket, freeing up space for potential new persistent flows. This enhancement introduces a modest trade-off: a slight reduction in processing speed and increased complexity in hardware deployment, including the need to specify additional parameters, such as the total number of windows, before deployment. Importantly, all operations in OPT-PALLAS are designed to operate within the limited arithmetic capabilities of programmable switches, ensuring practical deployability.

We rigorously model and analyze PALLAS and OPT-PALLAS, providing formal theoretical guarantees. We then implement a prototype in P4 on a production-grade Intel Tofino switch. Experimental results show that PALLAS and OPT-PALLAS support traffic data rates over 60× higher than the state-of-the-art method [12], while achieving 4.21% and 7.85% higher F1 scores in low-speed networks. Furthermore, PALLAS and OPT-PALLAS utilize only 8.5% and 9.7% of the switch's total resources on average, while maintaining low per-packet processing latency—highlighting their practical efficiency and effectiveness. We also evaluate PALLAS and OPT-PALLAS on a CPU platform against additional baselines, and the superiority of our approach remains consistent.

II. BACKGROUND AND RELATED WORK

In this section, we first provide formal definitions of the data stream and the persistent flow detection problem. We then review existing approaches and summarize their limitations, which collectively motivate our design.

A. Formal Definition

Data Stream: Consider a sequence $S = \langle e_1, e_2, \ldots, e_n \rangle$ representing a stream of incoming flows, where each flow e_i is described by $\langle \text{key}, \text{value} \rangle$. In a network monitoring scenario, the key could be the source–destination IP pair, while the value reflects metrics such as the flow's persistence over time.

Persistence: Let M denote the total number of non-overlapping time windows in a high-speed data stream. Each window W_j $(1 \leq j \leq M)$ is defined by a specific boundary criterion, such as a fixed number of packets (e.g., every 5,000 packets). For a given flow e_i, its persistence P_{e_i} increases by one whenever e_i appears at least once within W_j, irrespective of the frequency of its occurrences in that window.

Persistent Flow Detection: A flow e_i is classified as γ-persistent if it appears in enough time windows such that $P_{e_i} \geq \gamma M$, where $\gamma \in (0, 1]$ is a user-selected threshold and M denotes the total number of windows.

B. Related Work

Existing sketch-based approaches for persistent flow detection can be broadly classified into two categories: single-dimensional [6], [7], [12], [17] and dual-dimensional [4], [5], [16], [18].

1) Single-Dimensional: This category of methods relies exclusively on tracked flow persistence for replacement operations during the update process. Small-Space (SS) [19] uses sampling to track item persistence in a hash table, reducing space overhead to some extent. However, it still retains many non-persistent items, resulting in inefficient memory usage. To address this, PIE [7] represents each item using a Raptor code [20], storing the code instead of the raw ID to reduce memory consumption. Despite this improvement, PIE must still maintain codes for all items within each time window, even though most are not persistent. Furthermore, the encoding and decoding processes are computationally intensive, making PIE unsuitable for high-speed data streams. On-Off Sketch [6] employs an On/Off flag to track flow persistence and uses a simple swap operation to handle hash collisions during updates. However, its naive replacement strategy often results in persistent flows being erroneously evicted from buckets. Pyramid-based On-Off Sketch [17] improves memory efficiency by incorporating dynamically sized counters. Yet, its complex design renders it impractical for deployment on programmable switches. Pontus [12] introduces a novel metric, the collision decay flag, to enable precise persistence tracking. However, its reliance on frequent flag resets limits its scalability in high-flow-rate scenarios, as demonstrated in Section VIII-B.

2) Dual-Dimensional: These methods incorporate dual features during replacement operations to prevent persistent flows from being easily replaced under limited memory budgets [4], [5], [16]. For example, P-Sketch [5] considers the packet arrival frequency of each flow, while Stable-Sketch [4] introduces a novel feature, bucket stability, which accounts for flow variation within a bucket to offer greater protection to persistent flows. Pandora [16] observes that most persistent flows are rarely absent for long periods. Leveraging this insight, it assigns higher eviction probabilities to flows that have remained inactive for longer durations, thereby freeing up space for truly persistent flows. Tight-Sketch [18] leverages flow arrival patterns and applies different replacement strategies based on a flow's persistence level. If a flow's persistence value exceeds a predefined threshold, a dual-feature strategy is used, considering both frequency and arrival continuity. Otherwise, the replacement decision is based solely on the persistence value. This approach increases the likelihood that highly persistent flows remain in the sketch, preventing them from being easily evicted by a large number of non-persistent flows, leading to high detection accuracy.

3) Limitations of Existing Methods: While dual-dimensional methods generally achieve higher detection accuracy than single-dimensional approaches, they incur greater memory consumption due to the additional features, reducing overall memory efficiency. Their performance

may degrade as the number of persistent flows increases. Moreover, both single- and dual-dimensional methods rely on frequent flag resets for persistence tracking, limiting their effectiveness on programmable switches and in high-speed networks, similar to the limitations observed in Pontus [12]. This dependence on flags for duplicate removal is a fundamental limitation of existing methods, as they typically overlook the hardware constraints of programmable switches.

C. Summary

In a nutshell, existing methods for persistent flow detection encounter significant challenges when implemented on hardware-programmable switches in high-speed flow scenarios. To overcome these limitations, we propose a novel sketch, PALLAS, which achieves high accuracy even with constrained memory resources and high-speed traffic rates, as detailed in the following section.

III. PALLAS DESIGN

We present the design of the vanilla PALLAS, outlining its underlying principles, data structure, and core operations: update and query.

A. Design Philosophy

(i) To ensure compatibility with programmable switches, we eliminate the flag reset operation typically performed at the beginning of each time window. This reset requires coordination between the control plane and data plane, introducing delays that are unsuitable for high-speed traffic. Instead, we use a global counter to track the index of each arriving packet and compute the corresponding time window. Each bucket stores the index of the last time window in which its tracked flow appeared. By comparing this stored index with the current window index (see Section III-C for details), we can determine whether the flow has arrived in the current window, removing the need for explicit flag resets used in prior work.

(ii) To improve detection accuracy under skewed traffic distributions, we adopt a probability-based replacement strategy [5], which has demonstrated superior performance even under tight memory constraints [9], [12], [21], [22]. Given that the persistence value of a flow is typically much smaller than its frequency and considering the limited arithmetic capabilities of programmable switches, we employ a lightweight probabilistic decay approach rather than direct probabilistic replacement. This is because, even for a persistent flow, its relatively small persistence value makes it vulnerable to being incorrectly evicted when directly compared against the large number of packets from all flows.

B. Data Structure

Existing sketch data structures can be categorized as flat [5], [12], [16], [18] and hierarchical [17], [23], [24]. While the latter often achieve higher memory efficiency through complex variable-size counters, their intricate designs render them impractical for deployment on programmable switches. Thus, we adopt a flat structure for PALLAS.

Figure 1 shows the data structure of PALLAS, which consists of r rows, each with w columns (buckets). Each row is

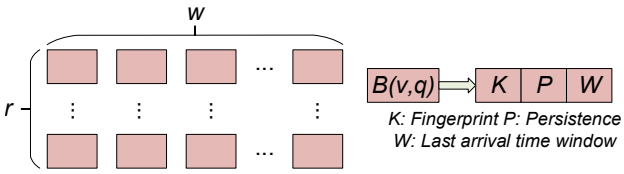


Fig. 1: Data structure of PALLAS.

associated with a unique hash function. A bucket $B(v, q)$ $(1 \leq v \leq r, 1 \leq q \leq w)$ contains three fields: the flow fingerprint (K), the persistence counter (P), and the last arrival time window (W). A global counter (C) is also maintained to record the current time window number. The fingerprint, a compact bit sequence derived from the flow key using the hash function, reduces memory usage and simplifies programmable switch implementation, as detailed in Section VI.

C. Main Operations

Update. Algorithm 1 illustrates the update process of PALLAS. Initially, all fields are initialized to 0 (Line 1). When a packet from a flow arrives, the global counter (C) increments by 1, and the current time window index is computed as $\lfloor C/G \rfloor + 1$, where G is the predefined time window size (Lines 2-4). Subsequently, one of three scenarios may occur:

(i) When a packet from a flow e_i arrives, PALLAS sequentially applies the hash functions $h_1, h_2, \ldots, h_r$ to locate the bucket $B(v, q)$ (Lines 5-7). If the bucket is empty, indicating no flow is currently tracked, the flow is inserted into the bucket. The fingerprint field is updated with the flow's fingerprint, the persistence counter is initialized to 1, and the last arrival time window is set to the index of the current time window. The hash operations terminate (Lines 8-10).

(ii) If the fingerprint stored in the hashed bucket matches the incoming flow's fingerprint, the algorithm compares the current time window with the tracked last arrival time window. If the two values differ (i.e., the last arrival time window is smaller), this indicates the flow's first appearance in the current time window. In this case, the persistence counter is incremented by 1, and the last arrival time window is updated to the current time window (Lines 11-15). Conversely, if the same flow arrives again within the same time window (i.e., the last arrival time window equals the current time window), the persistence counter remains unchanged. *This design eliminates the need for flags, allowing all operations to be fully implemented in the data plane.*

(iii) When collisions occur in every row, PALLAS chooses one row at random and applies the replacement in its hashed bucket (Lines 16–17). Unlike the approach of selecting the bucket with the smallest value [5], random selection eliminates the circular dependency introduced by comparisons, simplifying deployment on programmable switches. Next, the algorithm compares the current time window with the bucket's last recorded arrival time window to check if the tracked flow has already been processed in the current window. If it has, the incoming flow is discarded, and no replacement occurs (Lines 18-20). Otherwise, the tracked flow's persistence counter is probabilistically decremented, at a rate defined by

Algorithm 1: Update Process of Basic PALLAS

Input: a packet from flow e_i, hash functions $h_1, h_2, \ldots, h_r$, global counter C, time window size G, $\min_p \leftarrow +\infty$

1 **Initialization:** Set all bucket fields $K_{v,q}$ (fingerprint), $P_{v,q}$ (persistence), and $W_{v,q}$ (last arrival time window) to 0 for all buckets $B(v,q)$. Initialize global counter $C \leftarrow 0$.
2 **Step 1: Update Global Counter.**
3 Increment global counter: $C \leftarrow C + 1$
4 Compute the current time window index: currentWindow $\leftarrow \lfloor C/G \rfloor + 1$
5 **Step 2: Locate Bucket.**
6 **for** $v = 1$ *to* r **do**
7 index $\leftarrow h_v(e_i$.key)
// Case 1: Empty bucket
8 **if** $K_{v,index} == 0$ **then**
9 $K_{v,\text{index}}, P_{v,\text{index}}, W_{v,\text{index}} \leftarrow e_i$.fingerprint, 1, currentWindow
10 **return**
// Case 2: Same flow already tracked
11 **else if** $K_{v,index} == e_i.fingerprint$ **then**
12 **if** $W_{v,index} < currentWindow$ **then**
13 $P_{v,\text{index}} \leftarrow P_{v,\text{index}} + 1$
14 $W_{v,\text{index}} \leftarrow$ currentWindow
15 **return**
16 randRow $\leftarrow$ randomInteger$(1, r)$
17 randIndex $\leftarrow h_{\text{randRow}}(e_i$.key)
18 **Step 3: Probabilistic Decay.**
19 **if** $W_{randRow,randIndex} == currentWindow$ **then**
20 **return** // Discard the incoming flow if bucket is fresh in current window
21 **if** $random(0,1) < \frac{1}{P_{randRow,randIndex}+1}$ **then**
22 $P_{\text{randRow,randIndex}} \leftarrow P_{\text{randRow,randIndex}} - 1$
23 **if** $P_{randRow,randIndex} == 0$ **then**
24 $K_{\text{randRow,randIndex}} \leftarrow e_i$.fingerprint
25 $P_{\text{randRow,randIndex}} \leftarrow 1$
26 $W_{\text{randRow,randIndex}} \leftarrow$ currentWindow
27 **return**

the reciprocal of its current counter plus one (Line 21). If the decrement succeeds (Line 22) and the counter reaches zero (Line 23), the incoming flow replaces the tracked flow. To this end, the new flow overwrites the fingerprint, resets the persistence counter to 1, and records the current time window as the last arrival window (Lines 24–26).

This approach ensures higher persistence flows are more likely to remain in buckets, thereby achieving high detection accuracy. Additionally, the simplicity of this method makes it highly suitable for deployment on programmable switches.

Query. To retrieve persistent flows, PALLAS first determines whether the given flow is tracked in the buckets by hashing it across the rows. If the flow is found, its persistence counter is checked against the predefined threshold γM. If the persistence is greater than or equal to the threshold, the flow is classified as persistent; otherwise, it is not.

D. Running Examples

To clarify the update procedure, we provide several running examples illustrated in Figure 2. Here, PALLAS is configured with two rows, each containing three buckets. The size of each time window is set to 500 packets. Assuming that 2,100 packets have already arrived, the index of the current time window is calculated as $\lfloor 2100/500 \rfloor + 1 = 5$.

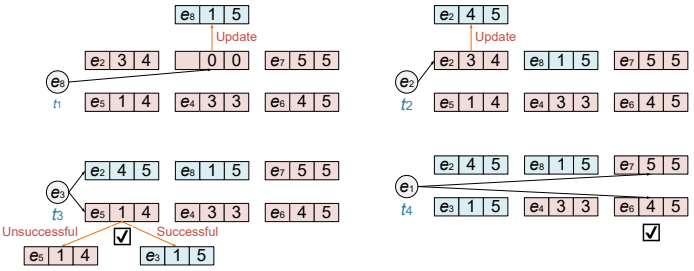


Fig. 2: Running examples of PALLAS.

At t_1, when the 2101st packet arrives, flow e_8 uses the hash function h_1 to locate a bucket in the first row. The bucket is empty: flow e_8 is inserted, the fingerprint is set to e_8's fingerprint, the persistence counter is initialized to 1, and the last arrival time window is set to the current time window (5).

At t_2, when e_2 arrives, it finds a matching fingerprint in the first row. PALLAS compares the tracked last arrival time window with the current time window. The last arrival window is 4, i.e., e_2 has not yet arrived in the current time window: the persistence counter is incremented by 1 (to 4), and the last arrival window is set to the current time window (5).

At t_3, when e_3 arrives, it encounters hash collisions across all rows. PALLAS randomly selects a bucket from one of the rows, i.e., the bucket tracking e_5. Since e_5's last arrival window is 4, i.e., it has not arrived in the current time window, a probabilistic decay is triggered with a probability of $\frac{1}{1+1}$. Two outcomes are possible: if the decay is unsuccessful, no changes occur and e_3 is discarded. If the decay succeeds, e_5's persistence counter is decremented by 1 to 0, letting e_3 replace e_5. The fingerprint is set to e_3's fingerprint, the persistence counter is set to 1, and the last arrival window is set to 5.

At t_4, when e_1 arrives, hash collisions occur across all rows, and the bucket tracking e_6 is randomly selected. However, because e_6's last arrival window matches the current time window (5), no replacement occurs, and e_1 is discarded.

IV. OPT-PALLAS: AN OPTIMIZED VERSION OF PALLAS

In the vanilla version of PALLAS, the probabilistic replacement mechanism triggered by hash collisions across all rows is based solely on flow persistence. However, as prior work has shown [5], [12], persistence values are typically much smaller than flow frequencies. Even a persistent flow–for instance, one observed across 1,600 time windows with a maximum persistence value of 1,600 [4]–can still be easily evicted from buckets due to the overwhelming volume of packets from other flows, making it more prone to incorrect replacement. Incorporating additional features beyond persistence can offer stronger protection for potential persistent flows, reducing their likelihood of being displaced by numerous non-persistent flows–particularly under highly skewed traffic and limited memory. To further enhance detection accuracy in such constrained settings, we propose OPT-PALLAS, an optimized version of PALLAS. OPT-PALLAS leverages the characteristic that most

persistent flows continue arriving without long interruptions, providing these flows with greater protection against eviction from buckets in skewed traffic. Note that these accuracy improvements come with a modest cost: a more sophisticated replacement strategy that slightly reduces processing speed and increases deployment complexity (discussed in Section IV-C).

A. Rationale

(i) If a flow has not arrived for an extended period, its likelihood of being truly persistent diminishes [5], [16]. Motivated by this insight, we enhance detection accuracy by offering stronger protection to flows that exhibit fewer arrival interruptions. The design of PALLAS naturally supports this strategy: each bucket records the last arrival window index W, while the current window index is derived from a global counter C. This allows us to estimate the number of inactive windows for a tracked flow. As this inactivity gap increases, we raise the probability of evicting the flow to free up space for more likely persistent flows.

(ii) To moderate the decay rate for persistent flows under skewed traffic, we adopt an exponential-based decay strategy considering both flow persistence and its arrival continuity. This approach makes such flows more resistant to eviction, even under severe hash collisions and limited memory, thereby improving detection accuracy.

B. Operations of OPT-PALLAS

1) Update: The cases where an incoming flow either finds an empty bucket or a matching bucket are handled identically to cases (i) and (ii) in the vanilla version of PALLAS, as described in Section III-C. The key difference between OPT-PALLAS and the basic PALLAS arises when hash collisions occur in all rows. Algorithm 2 illustrates the update process in such scenarios. When a flow collides in all rows, a row is randomly selected from the r rows in the sketch (Lines 1–2). If the tracked flow in that row has not appeared in the current time window, the incoming flow attempts a probabilistic decay (Lines 3–4). The algorithm first computes the inactivity, defined as the number of time windows since the tracked flow last appeared (Line 5). It then calculates the decay probability as $1 - e^{-\frac{\alpha \times inactivity}{P}}$, where α is a small positive constant that controls the decay speed (e.g., 0.11, see Section IX-C for more details), and P is the persistence of the tracked flow (Line 6). In this way, a flow with higher persistence and lower inactivity is more likely to be truly persistent, and thus has a lower probability of being decayed. This approach not only increases the likelihood of retaining genuine persistent flows in the sketch, but also enhances the accuracy of persistence tracking. A tracked flow is evicted from the bucket only when its persistence counter is decremented to zero (Lines 7–12).

2) Query: The query operation in OPT-PALLAS remains identical to that of the basic PALLAS.

C. Discussion

While the finer-grained control in OPT-PALLAS enhances the accuracy of persistent flow detection (as shown in Sections VIII-B and IX-A), it also introduces additional complexity. Specifically, the use of exponential-based computations poses a challenge for hardware implementation on the Tofino switch, which has limited support for complex mathematical operations. In addition, the constrained memory resources make it difficult to store precomputed values, as the probabilistic calculations involve numerous combinations of parameters, leading to substantial storage overhead. We detail these challenges in Section VI-B, where we also present a prototype implementation of OPT-PALLAS on the Tofino using P4 and runtime-assisted lookup tables to approximate exponential decay.

Algorithm 2: Update Process of OPT-PALLAS Under All-Row Hash Collisions

```
1  randRow ← randomInteger(1, r)
2  randIndex ← h_randRow(e_i.key)
3  if W_randRow,randIndex == currentWindow then
4      return
5  inactivity = max(currentWindow − W_randRow,randIndex, 1)
6  if random(0,1) < 1 − e^(−(α × inactivity)/P_randRow,randIndex) then
7      P_randRow,randIndex ← P_randRow,randIndex − 1
8      if P_randRow,randIndex == 0 then
9          K_randRow,randIndex ← e_i.fingerprint
10         P_randRow,randIndex ← 1
11         W_randRow,randIndex ← currentWindow
12 return
```

V. MATHEMATICAL ANALYSIS

In this section, we first demonstrate that both PALLAS and OPT-PALLAS do not overestimate persistence and subsequently derive the corresponding error bounds.

Theorem V.1 (No Over-Estimation Error)**.** *For any element $e_i \in \mathcal{S}$, its estimated persistence $\hat{P}_{e_i}$ produced by either Algorithm 1 or Algorithm 2 can not exceed its true persistence P_{e_i}. Formally, $\hat{P}_{e_i} \leq P_{e_i}$.*

Proof. We analyze the behavior of Algorithm 1 and Algorithm 2 over each time window W_j. For Algorithm 1, we employ a case analysis, while for Algorithm 2, we use a proof by contradiction. In both cases, we show that

$$\forall e_i \in \mathcal{S}, \forall j \in M, \hat{P}^j_{e_i} \leq P^j_{e_i}$$

Due to space limitations, the complete proof is provided in Section A of [25]. □

Here, we prove that both PALLAS and OPT-PALLAS achieve (ϵ, δ)-persistence, as defined in Section B of [25], thus demonstrating their low error rates in estimating persistent flows. Moreover, we show that OPT-PALLAS achieves a lower error rate than PALLAS, highlighting its superior estimation accuracy in identifying persistent flows.

Theorem V.2. *In Algorithm 1, given a small positive number ϵ, the probability of failing to identify a persistent flow is upper-bounded by $\delta = \frac{\lambda G + w(1-\exp(\Phi))}{\epsilon w \lambda G}$, where $\Phi = -\frac{(n-1)\lambda G}{2rw}$, w denotes the number of buckets in each row, r is*

the number of rows, G is the predefined time window size, n is the number of distinct flows, and λ is the arrival rate.

Proof. Due to space constraints, the detailed derivation of this bound is provided in Section C of [25]. □

Theorem V.3. *In Algorithm 2, given a small positive number ϵ, the probability of failing to identify a persistent flow is upper-bounded by $\delta_{opt} = \frac{\lambda G + w(1-\exp(\Phi))}{\epsilon w \lambda G}$, where $\Phi_{opt} = -\frac{\alpha(n-1)}{rw}$, w denotes the number of buckets in each row, r is the number of rows, G is the predefined time window size, n is the number of distinct flows, and λ is the arrival rate.*

Proof. Owing to space constraints, the detailed derivation of this bound is provided in Section D of [25]. □

Remark V.4. *Given that $\alpha < \frac{\lambda G}{2}$, we have*

$$\Phi_{opt} = -\frac{\alpha(n-1)}{rw} > -\frac{(n-1)\lambda G}{2rw} = \Phi.$$

Since the exponential function is strictly increasing, it follows that $e^{\Phi_{opt}} > e^{\Phi}$, and thus $1 - e^{\Phi_{opt}} < 1 - e^{\Phi}$. Substituting into the expressions for the error bounds yields

$$\delta_{opt} < \delta.$$

Therefore, OPT-PALLAS *provides a strictly smaller upper bound on the error probability than* PALLAS.

VI. TOFINO HARDWARE IMPLEMENTATION

We implement a prototype of PALLAS and OPT-PALLAS in a testbed with industry-grade Intel Tofino programmable switches using the P4 language.

A. PALLAS *Implementation*

Figure 3 shows the mapping of PALLAS onto a PISA pipeline. The figure depicts a toy example including scenarios of hash collision and no collision of a flow passing through the pipeline. The design of PALLAS in Tofino consists of three main components: the blocks per row and a final block responsible for determining whether to replace a flow or decrease its persistence count. We employ the Tofino Native Architecture (TNA) RegisterAction extern function to keep track of flows, their most recent arrival times, and their persistence counts. Register entries are allocated according to the number of buckets used in the experiments. To generate a unique key for each flow and store the flow keys in the flow tracking registers, we use the predefined hash function of `HashAlgorithm_t.CRC{X}`, applying different bit sizes for X. Each incoming packet is assigned a 32-bit unique flow key, which is then used to check for a match in the corresponding register index (i.e. bucket). The reason why we use 32-bit hash value instead of using total 64-bit source and destination ip addresses as a flow key is to reduce the memory to store the key and optimize the switch implementation. The optimization is necessary to break dependencies, as compound conditions are not allowed in P4 implementations and must be expressed through nested evaluations.

If the flow in the bucket differs from the incoming flow, a collision is detected. If a collision occurs in both rows, the packet is forwarded to the replacement decision block, as depicted in Figure 3, where the target bucket for replacement or decrement operations is determined using $RAND(0,1)$. After determining the target bucket, we recirculate the packet to eliminate dependencies, which increases the total number of stages required in the switch. Additionally, this approach is necessary because the TNA restricts accessing the same register more than once per packet.

In our experiments with both PALLAS and OPT-PALLAS, although the recirculation rate varies with memory allocation, it remains low enough to have no noticeable impact on system functionality, as the design efficiently handles traffic regardless of rate. In contrast, Pontus shows a lower recirculation rate but with a limited processing capacity of 5 Mbps, highlighting a trade-off between recirculation efficiency and throughput. Overall, the recirculations in both in PALLAS and OPT-PALLAS are manageable because it does not impede traffic flow or introduce significant latency–only adding a delay on the order of a few nanoseconds. This ensures that the system operates efficiently, even under varying load conditions and with potential hash collisions.

After recirculation, we discard the stored flow or decay its persistence counter if the condition $RAND(0, 2^b) \leq int(\frac{2^b}{1+P_{\text{randRow, randIndex}}})$ and $W_{randRow} < T_{current}$ holds, where b is the bit size of the total time windows, $W_{randRow}$ is the arrival time of the most recent packet of the flow, and $T_{current}$ is the global current time value. It is challenging to implement division operations in the switch due to hardware constraints and limitations of the P4 language. Although the `MathUnit` extern allows for approximate division, it doesn't produce exact results and is restricted by factors such as the approximation method and the range of inputs. To mitigate these limitations, we precompute more accurate division approximations offline and populate the corresponding entries in the switch's table.

If a flow match occurs in one of the rows, the persistence counter is incremented by one, provided it has not been incremented previously within the same window, as in Figure 3. Finally, all packets traversing the switch pipeline are forwarded.

B. OPT-PALLAS *Implementation*

1) Challenges: The deployment of OPT-PALLAS on programmable switches introduces several challenges. Firstly, these switches are highly constrained environments, particularly with regard to executing complex mathematical operations such as exponential computations. While the use of `MathUnit` externs can facilitate the computation of exponentials, they yield only approximate results and incur additional pipeline stage consumption. Given that programmable switches typically support a maximum of 12 pipeline stages, it is imperative to investigate more efficient and accurate approaches for performing such calculations. Thus, similarly to PALLAS, we implement decay probability calculations using

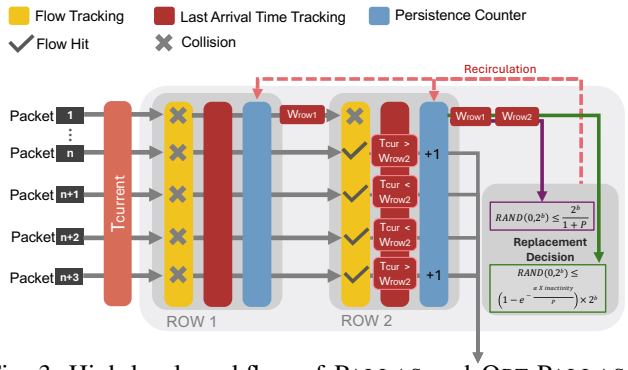


Fig. 3: High-level workflow of PALLAS and OPT-PALLAS in Tofino, illustrating scenarios of hash collision and no collision for a flow.

lookup tables in programmable switches. The probability value depends on two parameters: P, representing the persistence of the tracked flow, and *inactivity*, which denotes the number of time windows since the flow last appeared. This results in a total of $(window\ size)^2$ possible combinations. Given the limited resources available in programmable switches, it becomes necessary to compress the table to ensure the implementation remains feasible across various window sizes.

2) Implementation Details: The hardware deployment of PALLAS and OPT-PALLAS differ on how we take the replacement decision, as shown in Figure 3. Different from PALLAS, after recirculation, we discard the stored flow or decay its persistence counter if the condition of $RAND(0, 2^b) \leq int((1 - e^{-\frac{\alpha \times inactivity}{P_{\text{randRow,randIndex}}}}) * 2^b)$ and $W_{randRow,randIndex} < T_{current}$ holds. To support this, we define a lookup table that stores all possible combinations of (*inactivity*, $P_{\text{randRow,randIndex}}$), along with the corresponding precomputed values of the right-hand side of the inequality. Due to the high cost of storing all possible combinations—and the observation that different *inactivity* values can yield close results for the same $P_{\text{randRow,randIndex}}$—we compress the table entries by using ternary matches with appropriate masks on *inactivity*, combined with exact matches on $P_{\text{randRow,randIndex}}$. This means that pairs that share the same $P_{\text{randRow,randIndex}}$ and fall within the same *inactivity* range are mapped to a single precomputed value, which is computed as the average of all corresponding values within that range. With this mechanism, the table becomes scalable to arbitrary window sizes without degrading the performance of OPT-PALLAS. However, the window size must be known in advance to appropriately configure the lookup table compression.

VII. EVALUATION SETUP

Platforms. To evaluate the performance of PALLAS and OPT-PALLAS, we implement it on two platforms: *(i) Hardware*: A real testbed using off-the-shelf Intel Tofino switches programmed in P4. *(ii) Software*: A C++ software version running on a CPU platform equipped with an Intel(R) Core(TM) i5-1135G7 @ 2.40GHz processor and 16GB of DRAM.

Baselines. On the Tofino switch, we compare PALLAS and OPT-PALLAS with Pontus [12], the most state-of-the-art method for persistent flow detection. In the Tofino hardware implementation, PALLAS operates using two rows, while Pontus is constrained to a single row due to its more complex update mechanism. Both PALLAS and OPT-PALLAS use 32-bit fingerprints, resulting in negligible collision probability even under high-volume traffic and tight memory constraints. The number of buckets on the Tofino switch is varied from 1,024 to 16,384, following prior configurations [26]. Since PALLAS, OPT-PALLAS and Pontus employ identical bucket sizes, the total memory usage remains consistent for a given number of buckets, ensuring a fair and balanced comparison.

On the CPU platform, we further validate the consistent superiority of PALLAS and OPT-PALLAS by comparing them against additional recent baselines, including Pandora [16], Stable-Sketch [4], Tight-Sketch [18], and P-Sketch [5]. Although other methods such as On-Off Sketch [6] and PIE [7] exist, the selected baselines consistently outperform them, and thus we omit comparisons with those less competitive approaches. For a fair comparison, all baseline methods are adapted to use fingerprint-based flow tracking. We set the number of rows in PALLAS and OPT-PALLAS as 2 [5], [12]. The settings of all considered baselines are aligned with [12]. We vary the memory size from 16KB to 256KB [4], [12], [16], aligning with the typical range of L1/L2 cache sizes [27].

Traces. We evaluate PALLAS using three real-world CAIDA traces from 2016, 2018, and 2019. The 2016 and 2018 traces contain 22.3M packets from 730K and 760K flows, respectively, while the 2019 trace includes 29.5M packets from 1.53M flows. Each trace is split into 1,500 time windows, with the persistent threshold γ set to 0.4 [12]. Under this setting, the ground-truth number of persistent flows is 951, 1,075, and 1,886 for the 2016, 2018, and 2019 traces, respectively. To assess robustness, we also vary the number of windows and the threshold in Sections IX-B.

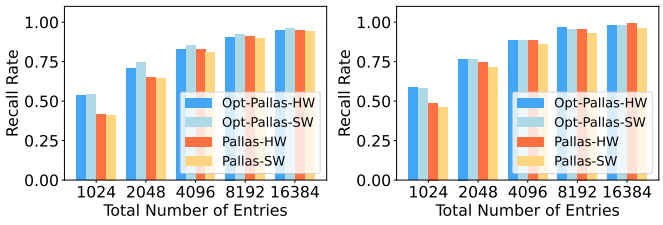
Parameter Settings. The basic PALLAS does not rely on any additional parameters. OPT-PALLAS introduces a decay probability controlled by a tunable parameter α, which adjusts the decay speed. We set α to a small positive value of 0.11. This parameter selection follows the common practice in prior sketch-based approaches [5], [16], [21], [22], where values are determined empirically. A detailed analysis of the impact of α is provided in Section IX-C.

Metrics. We evaluate performance using five metrics: (i) *Recall*, the ratio of correctly identified persistent flows to all true persistent flows; (ii) *Precision*, the ratio of correctly identified persistent flows to all reported persistent flows; (iii) *F1 score*, the harmonic mean of recall and precision; (iv) *AAE* (Average Absolute Error), which measures the average error between estimated and true persistence values; and (v) *Update throughput*, measured as the number of packets processed per second (Mpps). All experiments are repeated five times, and we report the average values.

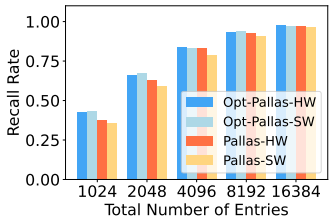
VIII. HARDWARE EVALUATION (TOFINO-BASED)

A. Alignment of Tofino and Software Versions

We first compare the recall for identifying real persistent flows on both the Tofino switch and the CPU platform across

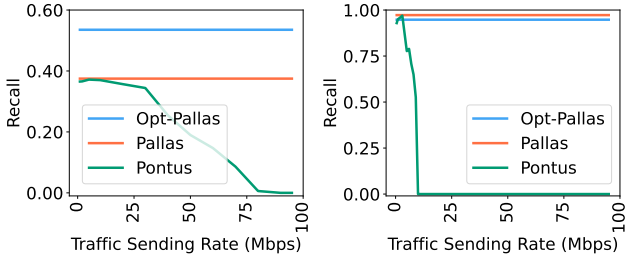


(a) CAIDA 2016. (b) CAIDA 2018.



(c) CAIDA 2019.

Fig. 4: Recall for PALLAS's hardware (Tofino) and software (CPU) implementations, showing consistent results and validating the accuracy of our hardware implementation (In the legend, HW and SW denote hardware and software).



(a) 1,024 buckets. (b) 16,384 buckets.

Fig. 5: Recall of OPT-PALLAS (2-row), PALLAS (2-row) and Pontus (1-row), both implemented in hardware, under different traffic sending rates.

different traces. *Since both methods only exhibit underestimation errors—meaning all captured persistent flows are genuine—the precision is always 1.* Hence, we omit the precision metric. As shown in Figure 4, the capture rates remain similar across various register entry sizes, demonstrating the correctness and consistency of our hardware implementation.

B. Comparative Evaluation in Tofino Hardware Switch

We first evaluate the detection accuracy of OPT-PALLAS, PALLAS and Pontus at varying traffic sending rates, using 1,024 and 16,384 buckets, shown in Figure 5. We observe that both OPT-PALLAS and PALLAS perform reliably even at high sending rates, and their performance remains unaffected by traffic speed. This is because PALLAS 's entire implementation resides in the data plane, requiring no control-plane interaction. In contrast, Pontus's frequent flag resets cause its recall to drop significantly once the traffic injection rate exceeds around $5-10$ Mbps, depending on the allocated memory size, rendering it less suitable for high-speed environments.

Figure 6 shows the recall and F1 score for OPT-PALLAS, PALLAS and Pontus, tested with a feasible traffic sending rate, on the hardware platform. As illustrated, OPT-PALLAS significantly outperforms PALLAS in all scenarios with limited memory resources, while OPT-PALLAS's performance becomes comparable—mostly slightly better—when more memory is allocated. However, on the CAIDA 2019 trace, OPT-PALLAS consistently outperforms PALLAS across all memory allocations. The maximum recall and F1 score improvement achieved by OPT-PALLAS over PALLAS is 11.77% and 10.82%, respectively, whereas the highest gain observed for PALLAS over OPT-PALLAS is only 3.1% and 0.71%. The superiority of OPT-PALLAS comes with certain limitations, such as the constraints on the number of entries that can be stored due to limited table capacity. While PALLAS allows us to store millions of entries in its table to calculate the probabilistic values, OPT-PALLAS allows to store data in only 25K entries. This is because, in PALLAS, the probabilistic value depends solely on the persistence value, allowing the use of a table with exact matches. In contrast, OPT-PALLAS computes the probability based on both persistence and *inactivity* values, requiring a more complex table that combines ternary and exact matches, which limits the memory we can allocate to the entries.

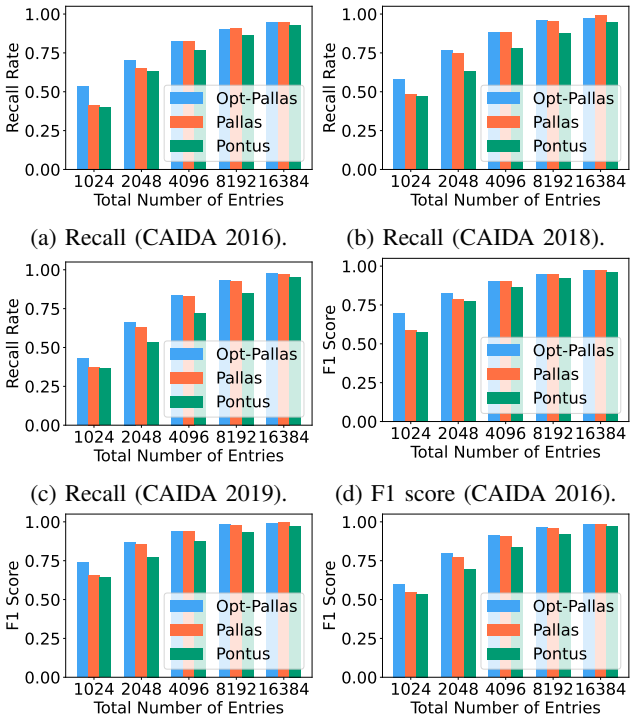
Moreover, both OPT-PALLAS and PALLAS consistently outperform Pontus in terms of accuracy. For instance, on the CAIDA 2018 trace, recall of OPT-PALLAS and PALLAS is on average 9.28% and 7.11% higher, respectively, while on the CAIDA 2019 trace, F1 score of OPT-PALLAS and PALLAS is on average 5.98% and 4.21% higher, respectively. This improvement stems from their probabilistic decay strategy, which prevents persistent flows from being easily evicted by non-persistent ones. Moreover, both of their elegant update procedure supports a 2-row deployment, further boosting detection accuracy.

Resource	OPT-PALLAS	PALLAS	Pontus (1-row)
Hash Bit	4.9%	5.7%	5.7%
Match Crossbars	5.7%	6.0%	4.6%
Gateways	16.7%	17.2%	16.7%
Logical Table ID	23.4%	23.4%	21.9%
VLIW Instruction	7.8%	7.3%	7.3%
SRAM	3.3%	4.3%	4.3%
TCAM	13.9%	0%	0%
Stages Used	12	12	11
Total Average	9.7%	8.5%	8.2%

TABLE I: Comparison of Resource Usage in Tofino Switch

C. Resource Usage and Latency

We examine resource usage, which is an important parameter to consider when dealing with resource-constrained user plane devices, and latency, using the Intel P4 Insight analysis tool [28] offering an in-depth analysis of compiled P4 programs. Table I shows the resource usage of OPT-PALLAS, PALLAS, and Pontus in Tofino Switch. OPT-PALLAS uses 9.7% of total resources, which gives enough space for other operations of the switch. OPT-PALLAS and PALLAS utilize all the stages available because of the dependencies between the implementation blocks shown in Figure 3. Both OPT-PALLAS and PALLAS's hardware design is centered around registers that consume SRAM resources, yet they use only



(a) Recall (CAIDA 2016). (b) Recall (CAIDA 2018).

(c) Recall (CAIDA 2019). (d) F1 score (CAIDA 2016).

(e) F1 score (CAIDA 2018). (f) F1 score (CAIDA 2019).

Fig. 6: Recall and F1 score of feasible hardware implementations of OPT-PALLAS (2-row), PALLAS (2-row) and Pontus (1-row). PALLAS consistently achieves higher accuracy than Pontus across all evaluated scenarios.

3.3% and 4.3% of the total available SRAM, respectively. While OPT-PALLAS utilizes less SRAM, it requires TCAM resources (13.9%) to implement ternary matches different than PALLAS and Pontus. Overall, in most of the resources, OPT-PALLAS consumes less. Despite of the 2-row implementation in OPT-PALLAS and PALLAS, they have generally a similar resource usage with Pontus.

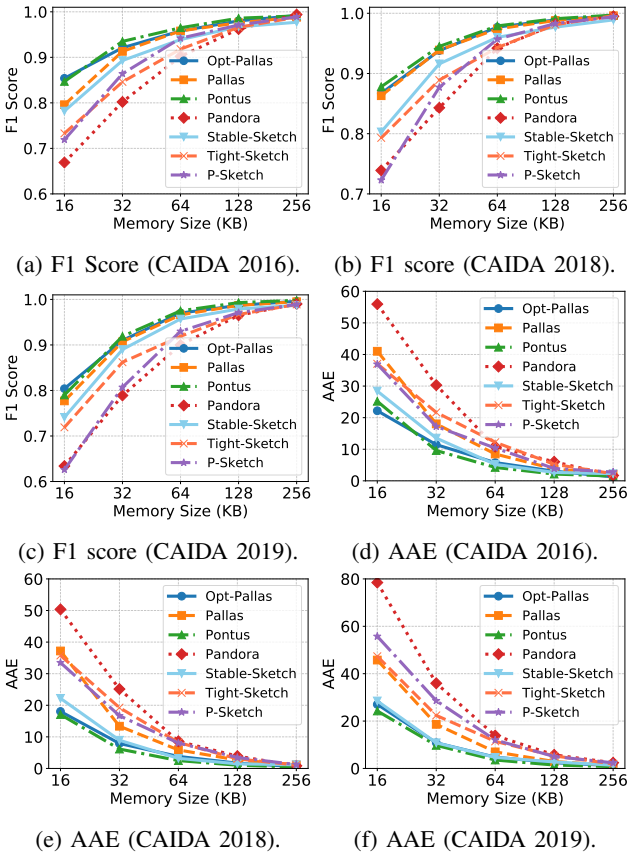
Besides utilizing a reasonable amount of memory, OPT-PALLAS detects persistent flows at line rate, introducing an average packet processing latency of just 443 ns. This represents only a 15 ns and 34 ns increase over the PALLAS and hardware deployable version of Pontus with 1-row, respectively, and a 132 ns increase compared to legacy forwarding in the switch.

IX. TRACE-DRIVEN EVALUATION (CPU-BASED)

A. Detection Performance

1) Accuracy: Figure 7 presents the detection accuracy of various methods. From Figures 7(a)–(c), we highlight three key observations: (i) OPT-PALLAS consistently achieves higher F1 scores than PALLAS. For example, with 16KB of memory, OPT-PALLAS outperforms PALLAS by 7.29% on the CAIDA 2016 trace. On average, OPT-PALLAS maintains superior F1 scores across all traces, attributed to its ability to incorporate arrival patterns and better preserve persistent flows under limited memory and high hash collision rates. (ii) The F1 scores of OPT-PALLAS are comparable to those of the state-of-the-art Pontus (with 2-row). While Pontus benefits from multiple flags for finer-grained persistence tracking, its reliance on frequent flag resets and complex update logic prevents practical deployment under a 2-row setting on the Tofino switch. (iii) OPT-PALLAS outperforms other baselines. Under the constrained 16KB memory setting, OPT-PALLAS improves the F1 score over Stable-Sketch by 9.21%, 7.97%, and 8.5% on the CAIDA 2016, 2018, and 2019 traces, respectively.

From Figures 7(d)–(f), we observe that OPT-PALLAS consistently yields significantly lower estimation errors compared to PALLAS across all traces. Additionally, the AAE of OPT-PALLAS is comparable to that of the advanced Pontus, which, despite its accuracy, is impractical for deployment on programmable switches. Compared to other baselines, OPT-PALLAS substantially reduces estimation errors, further validating its effectiveness.



(a) F1 Score (CAIDA 2016). (b) F1 score (CAIDA 2018).

(c) F1 score (CAIDA 2019). (d) AAE (CAIDA 2016).

(e) AAE (CAIDA 2018). (f) AAE (CAIDA 2019).

Fig. 7: F1 score and AAE of different methods for persistent flow detection, as a function of memory size.

2) Processing Speed: The results in Section VIII confirm that both PALLAS and OPT-PALLAS can handle packets at line rate on the Tofino switch. To further evaluate their performance, we measure their update speed on a CPU platform. Table II reports the average update speeds across memory sizes ranging from 16KB to 256KB. We observe that PALLAS achieves comparable speed to existing methods, while OPT-PALLAS exhibits slightly lower throughput due to its more complex replacement logic involving exponential computations. However, OPT-PALLAS still meets the performance requirements of high-speed networks such as 10Gbps, which require 14.88M packets per second processing speed [15].

Method	OPT-PALLAS	PALLAS	Pontus	Pandora	Stable-Sketch	Tight-Sketch	P-Sketch
CAIDA 2016	20.8	22.0	23.1	22.2	22.1	21.6	21.9
CAIDA 2018	22.2	23.0	24.0	22.9	23.2	22.7	23.0
CAIDA 2019	21.9	22.7	23.9	23.1	22.9	22.4	22.7

TABLE II: Processing speed (Mpps) comparison across methods and traces.

α	0.01	0.03	0.05	0.07	0.08	0.09	0.10	0.11	0.12	0.14	0.16	0.40	0.50
CAIDA 2016	0.762	0.813	0.834	0.847	0.845	0.847	0.849	0.854	0.849	0.850	0.798	0.802	0.787
CAIDA 2018	0.796	0.848	0.856	0.864	0.858	0.868	0.880	0.867	0.874	0.868	0.866	0.856	0.853
CAIDA 2019	0.727	0.769	0.784	0.783	0.799	0.792	0.798	0.804	0.795	0.794	0.798	0.786	0.776
Average	0.762	0.810	0.825	0.831	0.834	0.836	0.842	0.842	0.839	0.837	0.821	0.815	0.805

TABLE III: F1 Score under different α values (memory = 16KB).

B. Multiple Cases

1) Performance under Different Thresholds: In the previous evaluation, the persistence threshold γ was set to 0.4. To validate the robustness of our approach, we vary γ from 0.1 to 0.7, using a fixed memory budget of 16KB and the CAIDA 2019 trace for testing. As illustrated in Figure 8, OPT-PALLAS consistently achieves the highest F1 score across all threshold settings compared to the evaluated baselines. When the threshold is low (e.g., $\gamma = 0.1$), the number of identified persistent flows increases, and the distinction between persistent and non-persistent flows becomes less clear, resulting in reduced detection accuracy. Even under this challenging condition, OPT-PALLAS outperforms Pontus by 4.2%, demonstrating its robustness and effectiveness.

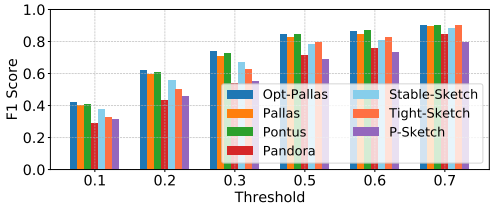


Fig. 8: Detection accuracy with varying thresholds.

2) Performance under Different Number of Time Windows: We vary the number of time windows from 1,000 to 2,000 using a 16KB memory under the CAIDA 2019 trace, with the persistent threshold $\gamma = 0.4$. As shown in Figure 9, OPT-PALLAS consistently outperforms the baselines across different window settings. For example, at 1,200 windows, OPT-PALLAS achieves an F1 score 2.37% higher than Pontus, demonstrating its effectiveness and robustness.

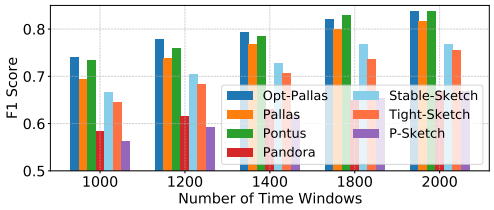


Fig. 9: Detection accuracy with varying time windows.

C. Parameter Analysis

In OPT-PALLAS, the parameter α controls the replacement speed of a tracked flow when hash collisions occur. A higher value of α increases the probability of replacement, which means that even flows with high persistence may face a higher risk of eviction. In contrast, a lower α slows down the replacement process, making it more difficult to evict even flows with low persistence. To select an appropriate value for α, we conduct empirical evaluations using various traffic traces. We vary α from 0.01 to 0.5 and fix the memory size at 16KB, an extreme case chosen to evaluate the performance of our method under highly constrained memory conditions.

As shown in Table III, the F1 scores across different traces exhibit an increasing trend as α increases from 0.01 to 0.11. However, further increases in α lead to a decline in performance. To achieve a high detection accuracy, we select $\alpha = 0.11$ as the default setting.

X. CONCLUSION

In this paper, we present PALLAS, a novel sketch for persistent flow detection fully implemented on the data plane of hardware programmable switches, enabling efficient performance in high-speed traffic scenarios. Unlike conventional methods, PALLAS eliminates the reliance on persistence-tracking flags, which require frequent control plane interactions and are unsuitable for fast traffic rates. Instead, PALLAS leverages probabilistic decay to accurately track persistent flows within buckets, even in skewed data distributions. Additionally, we propose an optimized version, OPT-PALLAS, designed to further enhance detection accuracy by considering persistent flow arrival patterns. We formally prove the correctness and soundness of both PALLAS and OPT-PALLAS and perform extensive evaluations on a Tofino hardware switch, demonstrating that both methods can handle high-speed data rates while offering superior detection accuracy. Additional experiments on a CPU platform further highlight the superior performance of OPT-PALLAS compared to existing state-of-the-art approaches.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/V038699/1, SNS JU and the European Union's Horizon Europe research and innovation program under Grant Agreement No. 101139270 (ORIGAMI) and No. 101093006 (TaRDIS). Beyza Bütün is supported by a Comunidad de Madrid predoctoral fellowship (PIPF-2022/COM-24867). Weihe Li was partially supported by Cisco through the Cisco University Research Program Fund (Grant No. 2019-197006).

REFERENCES

[1] Adel Alshamrani, Sowmya Myneni, Ankur Chowdhary, and Dijiang Huang, "A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1851–1877, 2019.

[2] Keval Doshi, Yasin Yilmaz, and Suleyman Uludag, "Timely detection and mitigation of stealthy ddos attacks via iot networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2164–2176, 2021.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[4] Weihe Li and Paul Patras, "Stable-sketch: A versatile sketch for accurate, fast, web-scale data stream processing," in *Proceedings of the ACM on Web Conference 2024*, 2024, pp. 4227–4238.

[5] Weihe Li and Paul Patras, "P-sketch: A fast and accurate sketch for persistent item lookup," *IEEE/ACM Transactions on Networking*, 2023.

[6] Yinda Zhang, Jinyang Li, Yutian Lei, Tong Yang, Zhetao Li, Gong Zhang, and Bin Cui, "On-off sketch: A fast and accurate sketch on persistence," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 128–140, 2020.

[7] Haipeng Dai, Muhammad Shahzad, Alex X Liu, Meng Li, Yuankun Zhong, and Guihai Chen, "Identifying and estimating persistent items in data streams," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2429–2442, 2018.

[8] Graham Cormode and Shan Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[9] Jiawei Huang, Wenlu Zhang, Yijun Li, Lin Li, Zhaoyi Li, Jin Ye, and Jianxin Wang, "Chainsketch: An efficient and accurate sketch for heavy flow detection," *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 738–753, 2022.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[11] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang, "Cocosketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 207–222.

[12] Weihe Li, Zukai Li, Beyza Bütün, Alec Diallo, Marco Fiore, and Paul Patras, "Pontus: A memory-efficient and high-accuracy approach for persistence-based item lookup in high-velocity data streams," in *Proceedings of the ACM on Web Conference 2025*, 2025.

[13] Marcelo C Luizelli, Ronaldo Canofre, Arthur F Lorenzon, Fábio D Rossi, Weverton Cordeiro, and Oscar M Caicedo, "In-network neural networks: Challenges and opportunities for innovation," *IEEE Network*, vol. 35, no. 6, pp. 68–74, 2021.

[14] Sajy Khashab, Alon Rashelbach, and Mark Silberstein, "Multitenant {In-Network} acceleration with {SwitchVM}," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 691–708.

[15] Lu Tang, Qun Huang, and Patrick PC Lee, "A fast and compact invertible sketch for network-wide heavy flow detection," *IEEE/ACM Transactions on Networking*, vol. 28, no. 5, pp. 2350–2363, 2020.

[16] Weihe Li, "Pandora: An efficient and rapid solution for persistence-based tasks in high-speed data streams," *Proceedings of the ACM on Management of Data*, vol. 3, no. 1, pp. 1–26, 2025.

[17] Yuanpeng Li, Xiang Yu, Yilong Yang, Yang Zhou, Tong Yang, Zhuo Ma, and Shigang Chen, "Pyramid family: Generic frameworks for accurate and fast flow size measurement," *IEEE/ACM Transactions on Networking*, vol. 30, no. 2, pp. 586–600, 2021.

[18] Weihe Li and Paul Patras, "Tight-sketch: A high-performance sketch for heavy item-oriented data stream mining with limited memory size," in *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, 2023, pp. 1328–1337.

[19] Bibudh Lahiri, Jaideep Chandrashekar, and Srikanta Tirthapura, "Space-efficient tracking of persistent items in a massive data stream," in *Proceedings of the 5th ACM international conference on Distributed event-based system*, 2011, pp. 255–266.

[20] Amin Shokrollahi, "Raptor codes," *IEEE transactions on information theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[21] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li, "Heavykeeper: an accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[22] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li, "Heavyguardian: Separate and guard hot items in data streams," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2584–2593.

[23] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui, "Stingy sketch: a sketch framework for accurate and fast frequency estimation," *Proceedings of the VLDB Endowment*, vol. 15, no. 7, pp. 1426–1438, 2022.

[24] Guoju Gao, Zhaorong Qian, He Huang, Yu-E Sun, and Yang Du, "Tailoredsketch: A fast and adaptive sketch for efficient per-flow size measurement," *IEEE Transactions on Network Science and Engineering*, 2024.

[25] Pallas, "Formal analysis," https://github.com/Mobile-Intelligence-Lab/Pallas, Accessed: 2025-08-11.

[26] Minjin Tang, Mei Wen, Junzhong Shen, Xiaolei Zhao, and Chunyuan Zhang, "Towards memory-efficient streaming processing with counter-cascading sketching on fpga," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[27] Simon Scherrer, Jo Vliegen, Arish Sateesan, Hsu-Chun Hsiao, Nele Mentens, and Adrian Perrig, "Albus: A probabilistic monitoring algorithm to counter burst-flood attacks," in *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2023, pp. 162–172.

[28] Intel, "P4 Insight," https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-insight.html.