ARMHH: Accurate, Rapid and Memory-Efficient Heavy Hitter Detection with Sliding Window in the Software-Defined Network Context

Zijun Hang*, Yongjie Wang† and Shuguang Huang‡
National University of Defense Technology
Hefei, China
Email: *hangzijun17@nudt.edu.cn, †w_yong_j@189.cn, ‡huangshuguang@ustc.edu

***Abstract*—Network measurement is essential to many network applications. The trend of ever-increasing network traffic calls for contemporary network measurement approaches to have three key characteristics: accuracy, timeliness, and memory efficiency. The sketch algorithm offers a suitable trade-off between accuracy and memory consumption; however, its interval method is not as accurate and rapid as the sliding window method in Heavy Hitter detection. However, the sliding window method is believed to be highly memory-consuming. Accordingly, this paper proposes ARMHH: Accurate, Rapid, and Memory-efficient Heavy Hitter detection, which combines the sliding window and sketch algorithm approaches. Compared with the interval method, our experiments show that ARMHH improves the accuracy by 121.8x, 2.8x, and 3.5x on average for CM, CU, and Hashpipe, respectively, by consuming additional memory overhead for the sliding window. ARMHH also compresses the sliding window memory by at most 60%.**

***Index Terms*—Heavy Hitter detection, Sliding window, Network measurement and analysis, Software-Defined Network**

I. Introduction

Network measurement is essential to many network applications, including anomaly detection [1], congestion control [2], [3], and billing [4]. The trend of ever-increasing network traffic results in the need for three key properties in network measurement: **accuracy**, **timeliness**, and **memory efficiency**.

However, measuring network traffic with full accuracy can result in high memory consumption. A network flow can be distinguished by its flow key, which is usually a 5-tuple (comprising source and destination IP addresses, source and destination ports, and protocol) in IP networks. The memory resources are excessively used if we measure each flow with full accuracy.

To achieve memory efficiency, many existing sampling methods [5]–[9] rely on probabilistic sampling the packets in the stream, which inevitably reduces accuracy.

Sketch algorithms [10]–[18] are developed in response to the desire to ensure high accuracy and low memory overhead in network measurement, which result in trade-offs of accuracy and memory consumption. A sketch algorithm allows different flows in the stream to share the same memory slot while extracting the statistics with guaranteed probability and provable accuracy loss. The emerging *Software-Defined Network* (**SDN**) [19] enables sketch algorithms to be deployed in network devices owing to its flexible data plane.

Heavy Hitter (**HH**) detection is a significant function within the sketch algorithms such as *Count-Min* (**CM**) sketch [16], Count-Min sketch with *Conservative Update* (**CU** Sketch) [17], and **Hashpipe** [18]. HH detection is designed to detect the flows whose traffic volume exceeds a given threshold percentage of the total traffic volume in the stream.

While traditional HH detection algorithms [16]–[18] address memory space and accuracy requirements simultaneously, they are still not accurate and rapid enough at finding HHs, because of the *interval method* (**INT**), which collects flow statistics from a given time interval, while only analyzing the HHs at the end of the interval. The chief drawback of INT is that it may fail to detect the HHs that appear on the boundaries of different intervals. Moreover, the time span of the interval can be reasonably large, leading to a delay in finding HHs.

The sliding window method is a novel direction that presents an alternative to the traditional INT and is more accurate and rapid in HH detection. Firstly, the sliding window method detects HHs continuously, which eliminates the HH loss in the interval boundaries. Moreover, the sliding window method can report large flows at any time during the interval, meaning that HHs can be reported at real time.

Notably, the sliding window method can be highly memory-consuming, which violates the original intention of sketch algorithms (namely to save memory).

Accordingly, this paper proposes **ARMHH**: ***A**ccurate, **R**apid and **M**emory-efficient **H**eavy **H**itter* detection with sliding window in the SDN context. The main contributions of this paper are as follows:

- This paper proposes ARMHH, which introduces the sliding window method into sketch network measurement algorithms and elaborates on the framework of ARMHH.
- We partition the ARMHH framework workflows into two parts, namely the SDN control plane part and SDN data plane part, to accommodate the framework to SDN architecture.
- We compress the sliding window memory space in ARMHH to save the on-chip memory, which is typically limited on network devices.

ISBN 978-3-903176-39-3

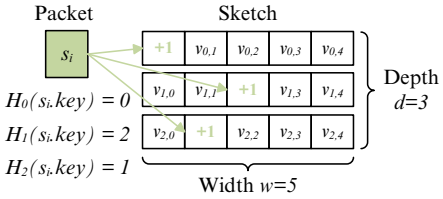


Fig. 1: Count-Min sketch overview.

TABLE I: Parameters in this paper.

Parameter	Description
S	packet stream with m packets, $S = \{s_0, s_1, \cdots, s_{m-1}\}$
h	length of the flow key (in byte)
g	length of the timestamp (in byte)
Φ	threshold of Heavy Hitters
t	time span of interval (in millisecond)
d	depth of the sketch
w	width of the sketch
$V_{d\times w}$	the sketch array
$H_i()$	hash function
M	memory overhead
$q = re$	depth of the sliding window
p	number of distinct flows in the sliding window
φ	packet repetition rate
L	set of HHs in S, $L = \{s_i \| f_{s_i} > \Phi q\}$

- We formulate the optimization problem of on-chip memory allocation for the sliding window and sketch algorithm.
- We demonstrate via experiments that the ARMHH is more accurate than the INT in terms of HH detection with the help of the sliding window.

II. Background and Motivations

A. Sketch algorithm

Sketch is a particularly powerful technique for real-time analysis of the massive, high-speed data generated by IP networks. Sketch is approximate and probabilistic, capable of answering a query within the error factor of ε with probability $1 - \delta$. For example, Figure 1 presents an overview of the Count-Min (CM) sketch [16]. CM sketch records the packets $\{s_0, s_1, \cdots, s_{m-1}\}$ of flow S in a $d \times w$ array $V_{d\times w}$, where d is the depth of the multi-level sketch and w is the width of the sketch. CM sketch records a packet s_i in d slots from the arrray, one in each level, by using d independent hash functions. Table I summarizes the parameters used in this paper.

B. Sliding window overview

Figure 2 illustrates the basic idea behind the sliding window method. The m packets $s_0, s_1, s_2, \cdots, s_{m-1}$ of stream S arrive sequentially. The sliding window is a First In First Out (FIFO) queue where each newly arriving packet enters the queue at the rear, while the outdated packets leave the queue at the front. The packets at the front of the sliding window

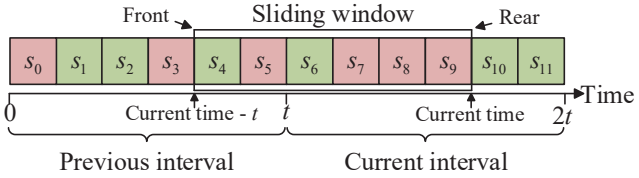


Fig. 2: The superiority of sliding window method against the traditional interval method in Heavy Hitter (HH) detection.

are examined every time when a new packet arrives and enters the sliding window at the rear. The front packets are expelled only if found to be outdated. This method behaves like a window sliding over the stream continuously, which is why it is referred to as the sliding window method.

C. Superiority of sliding window

The sliding window method is more accurate and faster than the interval method in HH detection. For example, in Figure 2, supposing that the span of the interval is t seconds and the threshold $\Phi = 0.3$, the $m = 12$ packets $\{s_0, s_1, \cdots, s_{11}\}$ of stream S arrive in two sequential intervals $[0, t)$ and $[t, 2t)$ at a constant rate of $\frac{6}{t}$ packets per second. Hence, both intervals include the same number of packets, namely six. For simplification, S only has two flows, the red-colored $S_0 = \{s_0, s_3, s_5, s_7, s_8, s_9\}$ and green-colored $S_1 = \{s_1, s_2, s_4, s_6, s_{10}, s_{11}\}$. The red-colored S_0 is a HH in reality, as the four packets $\{s_5, s_7, s_8, s_9\}$ arrive in a period shorter than t and are detected as a HH by the sliding window. However, in the interval method, S_0 is not recognized as a HH, as s_5 arrives in the previous interval $[0, t)$, while s_7, s_8, s_9 arrive in the current interval $[t, 2t)$; thus, S_0 is a false negative (F_N) in the interval method. The interval method naturally splits the whole stream S by epochs, meaning that the HHs at the boundary of intervals are neglected. We could decrease the threshold $\Phi = 0.25$ to detect the flow S_0 as a HH; however, this would mean that the flow S_1 is recognized as a HH as well, which is a false positive (F_P).

The sliding window method is more accurate than interval method in two key respects. Firstly, the sliding window method always counts the packets in the latest t seconds, i.e., its behavior involves sliding continuously over the stream, which eliminates the false negatives at the boundaries of intervals. Also, the HH is detected with a larger threshold Φ, which reduces the number of false positives.

Another advantage of the sliding window compared to the interval method is the speed with which it reports HHs. The sliding window method can report the HHs at any time, while the interval method can only report the HHs at the end of each interval. Notably, the time span of the interval can be reasonably large, leading to high latency in reporting HHs in the interval method. The sliding window always reports as soon as it finds HHs.

D. Challenges

SDN [19] has opened up a new era in sketch algorithms [11], [12], [18], [20] owing to its programmable data plane. However, to maintain the high throughput, the programmable data plane also places rigid restrictions on programming: (1) a small number of memory accesses per packet [21]; (2) no across-stage memory access for each packet [14]; and (3) simple operations for each packet. Moreover, the on-chip memory is scarce. For example, the Tofino programmable switch has only 16.3 MB of on-chip memory in total.

The main challenges of this paper are as follows: (1) modify the sliding window to satisfy the data plane operation restrictions; (2) reduce the memory overhead of the sliding window; and (3) integrate the sliding window with the sketch algorithm on the SDN architecture.

III. System Design

A. System overview

We introduce the sliding window method into the sketch algorithm to implement ARMHH. Figure 3 illustrates the framework of ARMHH, which consists of two parts: the data plane part and the control plane part. The data plane is good at processing the large volume of packets with a high throughput, but cannot carry out complicated operations like maintaining the HH set. By contrast, the control plane can carry out complicated operations but has a low throughput. Thus, the data plane part includes a sliding window and sketch algorithm that process every packet, while the control plane part maintains a set that records the key-size pairs of detected HHs.

The workflow of ARMHH presented in figure 3 comprises four steps: (1) update the sliding window, which maintains the packets that entered in the last t seconds; (2) update the sketch statistics according to the packets within the sliding window: insert the statistics of the new packet s_{q+k-1} into the sketch and expel the statistics of the outdated packet s_{k-1}; (3) the data plane reports any potential HH to the control plane (assuming the estimated size of s_{q+k-1} in the sketch is above the threshold, it sends $s_{q+k-1}.key$ and estimated size min_v to the control plane); and (4) the control plane updates the record in the HH set. Steps 1 and 2 run in the data plane, while steps 3 and 4 run in the control plane.

B. Basic version of sliding window

A basic version of the sliding window operates by directly storing the original timestamps and keys of packets that arrive in less than t seconds in a queue. Each packet accesses the rear element and possibly one or more front elements of the queue. When the flow rate is steady, one packet ejects only one front element in most cases; otherwise, we can eject a large number of front elements through the control plane. Thus, most packets only access the sliding window memory twice.

Definition 1 (sliding window depth q): $q > 0$ is the depth of the sliding window, i.e., the maximum number of packets that the sliding window queue needs to record in an interval.

The memory overhead for storing the original timestamps and original flow keys in a sliding window queue is as follows:

$$M_{sw}^{queue} = (g + h)q \tag{1}$$

where g is the timestamp width and h is the flow key width. However, M_{sw}^{queue} can be reasonably large. For example, when we use a six-byte integer as the timestamp and the 5-tuple as the flow key, where $g = 6$ and $h = 13$ bytes. On a 100 Gbps link, assuming the minimum packet size is 64 kB, the flow rate is 1.56×10^6 packets per second in the worst case. If we set the interval length to $t = 5$ milliseconds, the memory size of sliding window M_{sw}^{queue} is 148 kB, which is $8\times$ that of the sketch M_{sk} in our experiment.

C. Compressing the sliding window memory

In the basic sliding window scheme, M_{sw}^{queue} is positively correlated to the timestamp length g, flow key length h and window depth q. M_{sw}^{queue} can be large when g, h or q is large. Thus, we reduce the values of g, h and q to compress the sliding window memory.

1) Reducing the g and h: We can reduce the timestamp length g by using the precise length needed rather than a 6-byte. Also, we can reduce the flow key length h by storing the flow key hash values instead of the original flow keys.

Theorem 1: The memory overhead for storing the compressed timestamps and hash values of the flow keys in a sliding window queue is as follows:

$$M_{sw}^{queue'} = \frac{q}{8}\left(\left\lceil \log_2(t \times 10^3)\right\rceil + d\left\lceil \log_2 w\right\rceil\right) \tag{2}$$

Proof 1: Assuming the minimum unit of time is one millisecond, to represent a interval of t seconds, we will need $\left\lceil \log_2(t \times 10^3)\right\rceil$ binary bits. Thus, we need $\frac{1}{8}\left\lceil \log_2(t \times 10^3)\right\rceil$ bytes for each compressed timestamp. The length of the hash value is determined by the width of the sketch w. To index the w slots in each level of the d-level sketch will require $\lceil \log_2 w\rceil$ binary bits. Thus, we need $\frac{d}{8}\lceil \log_2 w\rceil$ bytes for the d indexes of each packet. The total amount of memory required for the sliding window therefore becomes $M_{sw}^{queue'} = \frac{q}{8}\left(\left\lceil \log_2(t \times 10^3)\right\rceil + d\left\lceil \log_2 w\right\rceil\right)$.

We have $M_{sw}^{queue} = (g + h)q$; letting $M_{sw}^{queue'} < M_{sw}^{queue}$, we have the corollary:

Corollary 1.1: In the sliding window queue, storing compressed timestamps and hash values of flow keys uses less memory than storing the original timestamp and flow keys, when the width of the sketch array w satisfies

$$w < 2^{\frac{1}{d}\left[8(g+h)-\left\lceil \log_2(t\times 10^3)\right\rceil\right]-1} \tag{3}$$

Example 1: For CM sketch, we usually adopt $d = 4$ or $d = 3$. Let $d = 4$, $t = 1$ and $h = 13$ in Equation 3; thus, we have $w < 2^{32}$. As long as the width of the sketch array w is below 2^{32}, this compression method is effective in this condition.

2) Reducing the q: We can also compress the window depth q by deleting the redundant keys. We can achieve this by replacing the queue with a linked list. Figure 4 presents

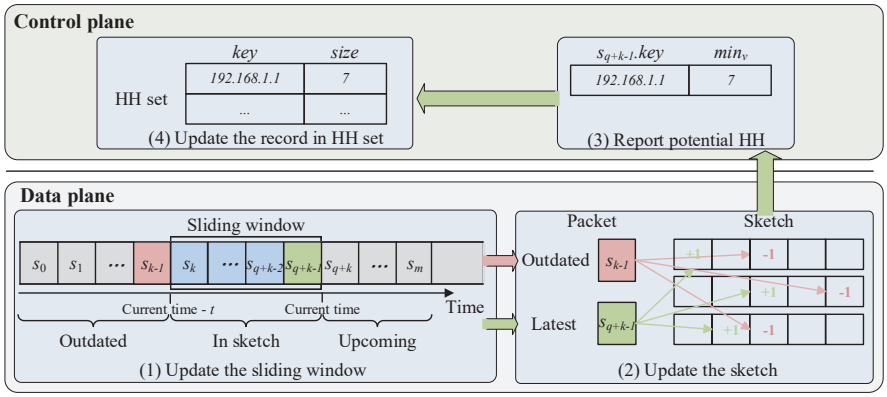


Fig. 3: Overview of the ARMHH framework.

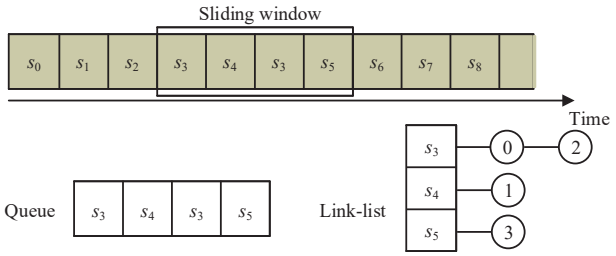


Fig. 4: The queue vs. linked list sliding window.

the data structure of a sliding window implemented in queue and linked list form respectively. The sliding window records four packets (s_3, s_4, s_3, s_5) where s_3 appears twice. The queue consumes one memory slot for each packet, meaning that the flow key of s_3 is repeated. By contrast, the linked list can merge the packets of the same flow by storing the distinct keys of the flows and the timestamps and indexes of packets that have the same flow key. In Figure 4, the packet s_3 is stored with its flow key, two timestamps and two indexes 0 and 2.

Definition 2 (distinct flows p): p is the number of distinct flows in the sliding window. We have $0 \leq p \leq q$.

Theorem 2: The memory overhead for storing the original timestamp and flow keys in a sliding window linked list is as follows:

$$M_{sw}^{list} = ph + \frac{q\lceil \log_2 q \rceil}{8} + gq \tag{4}$$

Proof 2: The p distinct flows consumes ph memory to store the p distinct flow keys. To index one of the q packets in the queue, we will use $\lceil \log_2(q) \rceil$ binary bits. The q packets will use $\frac{q\lceil \log_2(q) \rceil}{8}$ bytes to index all slots in the sliding window. The q packets use gq bytes to store the timestamp. Thus, $M_{sw}^{list} = ph + \frac{q\lceil \log_2(q) \rceil}{8} + gq$.

Example 2: In Figure 4, assume the window depth $q = 4$, timestamp length $g = 6$ bytes and the flow key length $h = 13$ bytes. The memory overhead for the queue $M_{sw}^{queue} = (g + h)q = (13 + 6) \times 4 = 76$ bytes. The memory overhead for the linked list $M_{sw}^{list} = ph + \frac{q}{8}(\lceil \log_2(q) \rceil) + gq = 3 \times 13 + \frac{4}{8}\log_2 4 + 6 \times 4 = 64$ bytes, which saves 12 bytes compared with M_{sw}^{queue}. Thus, the linked list uses less memory when there is a repeated flow. Consider a special situation, when all four packets have the same flow key: under these circumstances, M_{sw}^{list} is only 38 bytes.

Definition 3 (packet repetition rate φ): φ is the proportion of packets that have appeared before it in the sliding window.

$$\varphi = \frac{q - p}{q} = 1 - \frac{p}{q} \tag{5}$$

p is the number of distinct flows in Definition 2, while q is the depth of the queue in Definition 1. Thus, $q - p$ is the number of packets that have appeared before.

In Figure 4, $\varphi = 1 - \frac{3}{4} = \frac{1}{4}$. This means only one of the four packets (the s_3 between s_4 and s_5) has appeared before it in the sliding window. When $q = p$, all the flows appear only once and $\varphi = 0$. The smaller the p, the smaller the number of distinct flows and the higher the packet repetition rate φ. When all packets have the same flow key ($p = 1$), φ has the maximum value $1 - \frac{1}{q}$. (Note that $\varphi < 1$, as the front packet of the sliding window is always one that has never appeared before.) When the queue depth q is infinitely large and $p = 1$,

$$\lim_{q \to +\infty} \varphi_{max} = \lim_{q \to +\infty} \left(1 - \frac{1}{q}\right) = 1 \tag{6}$$

Moreover, $\varphi \geq 0$, as $q \geq p$ in Equation (5). Thus,

$$0 \leq \varphi \leq \varphi_{max} < 1 \tag{7}$$

Theorem 3: M_{sw}^{queue} and M_{sw}^{list} comply with

$$\begin{cases} M_{sw}^{queue} > M_{sw}^{list}, & \varphi > \frac{\lceil \log_2(q) \rceil}{8h} \\ M_{sw}^{queue} = M_{sw}^{list}, & \varphi = \frac{\lceil \log_2(q) \rceil}{8h} \\ M_{sw}^{queue} < M_{sw}^{list}, & \varphi < \frac{\lceil \log_2(q) \rceil}{8h} \end{cases} \tag{8}$$

Proof 3: We now prove that $M_{sw}^{queue} > M_{sw}^{list}$ when $\varphi > \frac{1}{8h}\lceil \log_2(q) \rceil$. Let $M_{sw}^{queue} > M_{sw}^{list}$, we have

$$hq + gq > ph + \frac{q\lceil \log_2 q \rceil}{8} + gq \tag{9}$$

Equation (9) $\iff h(q-p) > \frac{q\lceil \log_2 q \rceil}{8} \iff \frac{q-p}{q} > \frac{\lceil \log_2 q \rceil}{8h}$.

As $\varphi = \frac{q-p}{q}$ according to Definition 5, we have

$$\varphi > \frac{\lceil \log_2(q) \rceil}{8h} \tag{10}$$

We can also prove the equations of the other two conditions in Equation (8) as above.

According to Theorem 3, the linked list is more space-efficient than the queue only when the packet repetition rate φ is bigger than $\frac{\lceil \log_2(q) \rceil}{8h}$. Note that $\frac{\lceil \log_2(q) \rceil}{8h}$ is the ratio of the index length to the flow key length; the larger this ratio, the larger the minimum value of φ.

Example 3: Assuming $q = 1024$, $h = 13$ in Equation (10), we have $\varphi > 9.6\%$. Thus, as long as over 9.6% of packets appear more than once in the sliding window, the linked list will be more space-efficient than the queue.

3) Reducing h, g, and q: However, we still store the original flow keys in the linked list sliding window above, which only reduces q. We now reduce h, g, and q simultaneously by using a linked list sliding window that stores the compressed timestamps and hash values of flow keys.

Theorem 4: The memory overhead for storing the compressed timestamps and hash values of the flow keys in a sliding window linked list is as follows:

$$M_{sw}^{list'} = \frac{1}{8}\left(dp\lceil \log_2 w \rceil + q\lceil \log_2 q \rceil + q\left\lceil \log_2(t \times 10^3) \right\rceil\right) \tag{11}$$

Proof 4: We replace the original flow keys with the hash values of the flow keys and the original timestamps with compressed timestamps. Thus, we replace h with $\frac{d}{8}\lceil \log_2 w \rceil$ and g with $\frac{1}{8}\left\lceil \log_2(t \times 10^3) \right\rceil$ in Equation (4); accordingly, we have $M_{sw}^{list'} = p\frac{d}{8}\lceil \log_2 w \rceil + \frac{q}{8}\lceil \log_2 q \rceil + \frac{1}{8}\left\lceil \log_2(t \times 10^3) \right\rceil q$ $= \frac{1}{8}\left(dp\lceil \log_2 w \rceil + q\lceil \log_2 q \rceil + q\left\lceil \log_2(t \times 10^3) \right\rceil\right)$.

Also, we have the following:

Theorem 5: $M_{sw}^{queue'}$ and $M_{sw}^{list'}$ comply with

$$\begin{cases} M_{sw}^{queue'} > M_{sw}^{list'}, & \varphi > \frac{\lceil \log_2(q) \rceil}{d\lceil \log_2(w) \rceil} \\ M_{sw}^{queue'} = M_{sw}^{list'}, & \varphi = \frac{\lceil \log_2(q) \rceil}{d\lceil \log_2(w) \rceil} \\ M_{sw}^{queue'} < M_{sw}^{list'}, & \varphi < \frac{\lceil \log_2(q) \rceil}{d\lceil \log_2(w) \rceil} \end{cases} \tag{12}$$

Proof 5: Let $M_{sw}^{list'} < M_{sw}^{queue'}$; we thus have

$$\frac{dq}{8}\lceil \log_2 w \rceil < \frac{dp}{8}\lceil \log_2 w \rceil + \frac{q}{8}\lceil \log_2 q \rceil \tag{13}$$

$\iff \frac{q-p}{q} < \frac{\lceil \log_2 q \rceil}{d\lceil \log_2 w \rceil} \iff \varphi < \frac{\lceil \log_2 q \rceil}{d\lceil \log_2 w \rceil}$ We can also prove the equations of the other two conditions as above.

The above conclusion is in accordance with Theorem 3. Note that $\frac{\lceil \log_2(q) \rceil}{d\lceil \log_2(w) \rceil}$ is the ratio of the index length to the length of the d hash values.

D. Sliding window Heavy Hitter detection

We now integrate the sliding window with the sketch algorithm to detect HHs. We here adopt CM sketch and use a queue as sliding window as an example.

Algorithm 1 shows the transactions in the data plane, which include steps 1 and 2 in Figure 3. For each packet s_i in stream S, Algorithm 1 firstly inserts s_i into the CM sketch:

Procedure update($s_i.key$, min_v), update the sliding window Heavy Hitter set $L_{sw}^{'}$ in the control plane of the Software-Defined Network devices

Input: Flow key $s_i.key$ and estimated size min_v
Result: Updated sliding window Heavy Hitters $L_{sw}^{'}$

initialization: $L_{sw}^{'} \leftarrow \emptyset$;
if $\exists l_i^{'} \in L_{sw}^{'}$, **s.t.** $l_i^{'}.key == s_i.key$ **then**
  $l_i^{'}.size = \max(l_i^{'}.size, min_v)$; // update HH size
else
  $l_{new}^{'} \leftarrow$ pair $\langle s_i.key, min_v \rangle$;
  $L_{sw}^{'} = L_{sw}^{'} \cup \{l_{new}^{'}\}$; // insert s_i into HH set

Algorithm 1: Sliding window Heavy Hitter detection with CountMin algorithm in the data plane of Software Defined Network devices

Input: Packet stream $S = \{s_0, s_1, \cdots, s_{m-1}\}$
Result: Updated sliding window S_{sw}, and updated sketch value $V_{d\times w}$

1 initialization: $S_{sw} \leftarrow \emptyset$, hash functions $H_0, \cdots H_{d-1}$: $\{0, \cdots, 2^{8h}-1\} \rightarrow \{0, \cdots, w-1\}$, $V_{d\times w} \leftarrow \{0\}^{d\times w}$;
2 **for** $s_i \in S$ **do**
3 $index_{d\times 1} \leftarrow \{0\}^{d\times 1}$; // insert s_i into CM
4 **for** $j \leftarrow 0$ **to** $d-1$ **do**
5 $index_j \leftarrow H_j(s_i.key)$;
6 $k \leftarrow index_j$;
7 $v_{j,k} \leftarrow v_{j,k} + 1$;
8 $r_{rear} \leftarrow$ pair $\langle s_i.time, index_{d\times 1} \rangle$;
9 S_{sw}.pushback (r_{rear}) ; // push s_i to S_{sw} rear
10 $r_{front} \leftarrow S_{sw}$.front ();
11 **while** $r_{rear}.time - r_{front}.time \geq t$ **and** $!S_{sw}$.empty **do**
12 S_{sw}.popfront () ; // pop S_{sw} front
13 $idx_{d\times 1} \leftarrow r_{front}.index_{d\times 1}$;
14 **for** $j \leftarrow 0$ **to** $d-1$ **do**
15 $k \leftarrow idx_j$; // delete S_{sw} front in CM
16 $v_{j,k} \leftarrow v_{j,k} - 1$;
17 $r_{front} \leftarrow S_{sw}$.front ();
18 $min_v \leftarrow$ 0x7fffffff ; // estimate size of s_i
19 **for** $j \leftarrow 0$ **to** $d-1$ **do**
20 $min_v \leftarrow \min(min_v, v_{j,index_j})$;
21 **if** $min_v \geq \Phi q$ **then**
22 **call** update ($s_i.key$, min_v) ; // reprot HH

adds d slots in the d-level CM sketch array $v_{d\times w}$, where the j^{th} slot is indexed by $H_j(s_i.key)$, the j^{th} hash value of the flow key $s_i.key$ (Lines 3-7). Next, the timestamp $s_i.time$ and hash values $index_{d\times 1}$ of packet s_i are pushed to the rear of the sliding window S_{sw} (Lines 8-9). Moreover, Algorithm 1 deletes the outdated records at the front of the sliding window (Line 12), and decreases the corresponding slots in the CM sketch (Lines 13-16). Lastly, Algorithm 1 tracks the minimum value min_v of the d slots (Line 18-20) and reports it as a potential HH to the data plane if min_v exceeds $\Phi\,|S|$ (Lines 21-22).

Procedure update shows the transactions in the control plane. The control plane updates the HH set $L_{sw}^{'}$ according to

the flow key $s_i.key$ and estimated size min_v of the received potential HH. If the flow l'_i is already in L'_{sw} and $l'_i.size < min_v$, we update its size to min_v (Lines 2-3); otherwise, we insert the pair of $\langle s_i.key, min_v \rangle$ as a new HH into L'_{sw} (Lines 4-6).

We can also integrate the sliding window with CU sketch and Hashpipe with minor modification in Algorithm 1. CU uses the same sketch structure as CM; assuming each counter of sketch is a 4-bytes integer, the sketch memory for CU and CM is

$$M_{sk}(CU) = M_{sk}(CM) = 4dw \tag{14}$$

CU differs from CM only in terms of the way it increases and decreases the minimum slot across the d-level sketch. Hashpipe uses a different sketch structure from CM: in this case, we need to store the flow keys along with the sketch value in the multi-level sketch. Thus,

$$M_{sk}(Hashpipe) = (4 + h)dw \tag{15}$$

Hashpipe also differs from CM in terms of its sketch operation: more specifically, it may discard the prior flow's statistics when the prior flow collides with a new flow.

The biggest problem associated with integrating the sliding window with Hashpipe is as follows: we compressed the sliding window by only storing the hash indexes of the flow. If the prior flow is replaced with a new one, we may mistakenly decrease the statistics of the new flow when the packet of the prior flow leaves the sliding window. We evaluate the policy in section V-B1.

E. Query runtime

We can now query flow statistics in the control plane, including HH query and HH frequency query.

HH query. The set L'_{sw} in the control plane stores all potential HHs. Given a packet s_i, we need to look up whether $\exists\ l'_j \in L'_{sw}$ s.t. $l'_j.key == s_i.key$. If yes, s_i is regarded as a HH; otherwise, s_i is not a HH.

HH frequency query. We can also query the estimated frequency $\hat{f_{s_i}}$ of a packet s_i in the control plane. $\hat{f_{s_i}}$ is given by:

$$\hat{f_{s_i}} = \begin{cases} \frac{l'_j.size}{q}, & \textbf{if } \exists\ l'_j \in L' \textbf{ s.t. } l'_j.key == s_i.key \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

where q is the depth of the sliding window.

F. Formulating the optimization problem

We now formulate the optimization problem. Given the restrictions below:

$$\text{s.t.} \begin{cases} \varepsilon, \delta, \Phi > 0 \\ w, d, q, h, g \in \mathbb{N}^+ \\ p \le q,\ p \in \mathbb{N} \end{cases} \tag{17}$$

Our goal is to allocate minimum memory according to the value of φ:

$$min\ M = \begin{cases} M_{sk} + M_{sw}^{queue'}, & 0 \le \varphi \le \frac{\lceil \log_2(q) \rceil}{d \lceil \log_2(w) \rceil} \\ M_{sk} + M_{sw}^{list'}, & \frac{\lceil \log_2(q) \rceil}{d \lceil \log_2(w) \rceil} < \varphi < 1 \end{cases} \tag{18}$$

IV. Implementation

In this section, we implement a prototype of ARMHH, including a P4 data plane and Python control plane.

A. P4 data plane

We build the data plane in P4 (Programming Protocol-independent Packet Processors) [22], a language that specifies how programmable switches process packets.

We implement the sliding window and multi-level sketch using P4 registers. For the sliding window part, we use a register array as a round-robin queue that stores the compressed timestamps and hash indexes of packets. The data plane updates the queue driven by the arriving packets. Notably, the linked list structure is not well supported by the P4 platform; we may implement the linked list structure of the sliding window on other platforms in future works. For the sketch part, moreover, we use one register array for each level of the sketch value array $V_{d\times w}$ to store the flow statistics. We also need one additional flow key array $K_{d\times w}$ for Hashpipe to store the original flow keys in the multi-level sketch.

The data plane signals the control plane to eject front elements in the queue, when flow rate drops sharply. We run the P4 data plane on BMv2 (Behavioral Model version 2) [23], a software switch simulator that implements the packet-processing behavior specified by the P4 program.

B. Python control plane

We build a control plane in Python that receives the potential HHs from the data plane, updates the HH set and runs the query task answering the HH query and HH frequency estimation. Moreover, the control plane can also manipulate the sliding window queue according to the data plane's requests.

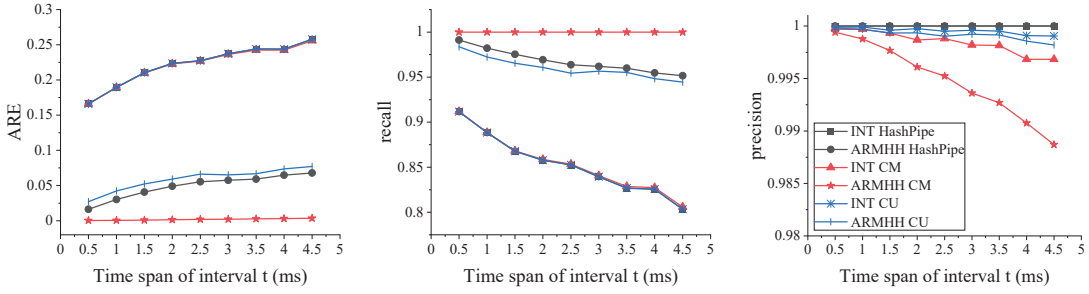
V. Evaluation

A. Experimental setup

1) Testbed: We build a simulation benchmark in the C++ language. We implement the CM, CU, and Hashpipe in both the *interval method* (**INT**) and ARMHH. We then evaluate them on an one-hour-long real network trace derived from the CAIDA 2018 dataset [24]. We use the 5-tuple as the flow key; thus, the length of the flow key h is 13 bytes. We split the one-hour-long trace into 3600 one-minute-long traces and compare the accuracy of INT and ARMHH at the end of each minute. The results are the average values from 3600 tests.

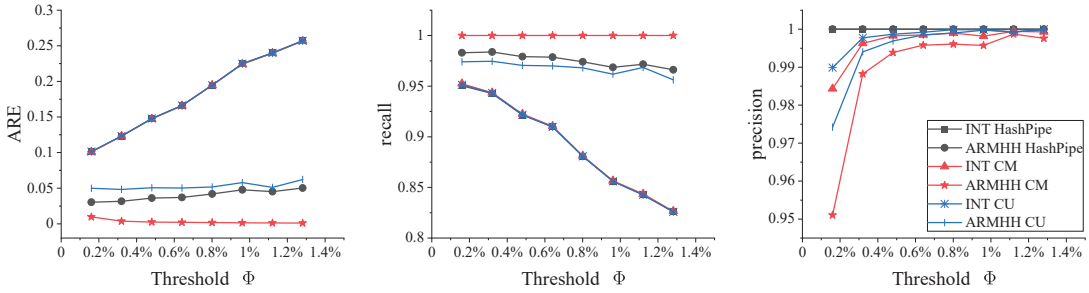
2) Metrics: We use the following evaluation metrics:

- Average Relative Error (ARE): $\frac{1}{|L|}\sum_{i=0}^{|L|-1}\frac{|f_{l_i}-\hat{f_{l_i}}|}{f_{l_i}}$.
- precision rate: $\frac{T_P}{|L'|}$, which is the ratio of true positives (T_P) among all positives $|L'|$ (i.e., the sum of true positives T_P and false positives F_P).



(a) ARE (b) recall (c) precision

Fig. 5: Accuracy under different time span of interval t.



(a) ARE (b) recall (c) precision

Fig. 6: Accuracy under different threshold Φ.

- recall rate: $\frac{T_P}{|L|}$ which is the ratio of true positives (T_P) among all real HHs $|L|$ (i.e., the sum of true positives T_P and false negatives F_N).

We use ARE for HH frequency query evaluation, while the precision and recall rate are used for HH query evaluation.

B. Accuracy comparison with same sketch memory

We first evaluate the accuracy of INT and ARMHH with the same sketch memory space, where the ARMHH consumes extra memory for the sliding window. We set $d = 3$, $w = 1500$ and use 4-bytes counter for the sketch, where $M_{sk}(CM) = M_{sk}(CU) = 17.6$ kB, and $M_{sk}(HashPipe) = 74.7$ kB.

1) Time span of interval t: We fix the threshold $\Phi = 0.001$ in order to examine the influence of the interval length t. Figure 5 presents the results when t varies from 0.5 to 4.5 milliseconds. As we have fixed a relatively small sketch memory size, the value of t cannot be too large; otherwise, the conflicts in the sketch will increase and degrade the accuracy.

Figure 5(a) reveals the ARE of the different algorithms. For CM, CU, and Hashpipe, ARMHH degrades the ARE from above 0.15 to under 0.07 compared with the INT under all values of t. As the time span t of the sliding window increases, the ARE of INT, ARMHH-CU, and ARMHH-Hashpipe increases linearly; however, the ARE of ARMHH-CM stabilizes. ARMHH benefits CM best among three algorithms in terms of ARE. The average improvements for CM, CU, and Hashpipe are $121.8\times$, $2.8\times$, and $3.5\times$, respectively.

Figure 5(b) plots the recall rate. ARMHH improves the recall rate on average for CM, CU, and Hashpipe by 15%, 11%, and 12%, respectively. As t increases, the recall rate decreases linearly in INT, ARMHH-CU, and ARMHH-Hashpipe. ARMHH best benefits CM, whose recall is always close to 100%. Part of the recall drop in ARMHH-Hashpipe is caused by the flow key compression, which mistakenly decreases sketch statistics (Section III-D). We also test Hashpipe with storing the original keys in the sliding window and the improvement is tiny: the ARE decreases by only 0.01 and recall increases by 0.015 when $t = 4.5$ ms.

Figure 5(c) shows the precision rate. The precision rate of ARMHH-Hashpipe is not affected by t and is always nearly to 100% in our experiment. The ARMHH reduces the recall rates of CM and CU because false negatives appear in the boundaries of different intervals. When $t = 4.5$ ms, the maximum recall rate loss is 0.8% in CM algorithm.

Accordingly, compared with the INT, ARMHH obtains better results in ARE and precision and acceptable loss in recall under all t.

2) Threshold Φ: We also fix $t = 2$ ms and vary the threshold Φ from 0.0016 to 0.012 in Figure 6. The smallest Φ_{min} is 0.0016; under this threshold, over 90% of all flows are extracted.

In Figure 6(a) and 6(b), it can be seen that the performance of ARMHH with all three algorithms is not affected by the varying Φ; However, the accuracy of INT degrades sharply when Φ gets larger. Figure 6(a) indicates that the ARE for INT reaches above 0.25 when Φ is 0.012; the HH frequency estimating results deviate largely from the actual results.

ARMHH improves the recall from below 0.83 to above 0.96 in the best case when $\Phi = 0.012$ in Figure 6(b). In Figure 6(c), the precision rates for INT and ARMHH are all above 0.99 when $\Phi \geq 0.0048$. The false negatives between different time intervals account for the accuracy loss in ARMHH. When $\Phi = \Phi_{min} = 0.0016$, the minimum precision is 95.1% in ARMHH-CM.

Accordingly, ARMHH achieves better results in terms of ARE and recall for all values of Φ, with minor loss in precision.

C. Quantifying memory overhead

We now quantify the sliding window memory overhead in ARMHH and compare it with the sketch memory overhead. The sliding window memory overhead M_{sw}^{queue}, $M_{sw}^{queue'}$, M_{sw}^{list}, and $M_{sw}^{list'}$ (Equations (1), (2), (4), and (11)) are related to the parameters p and q. In Figure 7, p and q are positively correlated with the time span of interval t; moreover, q increases much faster than p. Notably, $\varphi > \frac{\lceil \log_2(q) \rceil}{d\lceil \log_2(w) \rceil} > \frac{\lceil \log_2(q) \rceil}{8h}$ in Figure 7.

In Figure 8, the memory overheads for sliding window are linearly positively correlated with t. We have $M_{sw}^{queue} > M_{sw}^{list}$ as $\varphi > \frac{\lceil \log_2(q) \rceil}{8h}$ and $M_{sw}^{queue'} > M_{sw}^{list'}$ as $\varphi > \frac{\lceil \log_2(q) \rceil}{d\lceil \log_2(w) \rceil}$ for all t, which is in accordance with theorem 3 and theorem 5. The uncompressed solutions M_{sw}^{queue} and M_{sw}^{list} exceed $M_{sk}(CM) = 17.6$ kB at around $t = 1$ ms and $t = 1.5$ ms, respectively, while the compressed solutions $M_{sw}^{queue'}$ and $M_{sw}^{list'}$ exceed $M_{sk}(CM) = 17.6$ kB at around $t = 3.5$ ms and $t = 4.5$ ms, respectively. M_{sw}^{queue}, M_{sw}^{list}, $M_{sw}^{queue'}$ and $M_{sw}^{list'}$ are all smaller than $M_{sk}(Hashpipe) = 74.7$ kB for all values of t. We compress the sliding widow memory overhead by 70%-72% for the queue and 62%-68% for the linked list, respectively.

Accordingly, the sliding window memory overhead is linearly positively correlated with t, and we have further significantly compressed it. We can also conclude that the compressed queue and compressed list have comparable memory overhead; however, the compressed queue is easier to implement.

D. Tuning the sketch width

We now tune the width w of the sketch array. Figure 9 illustrates the ARE for CM algorithm with INT and ARMHH under t=1, 3 and 5 ms, fixing Φ to 0.01. Note that the x-axis of Figure 9 is logarithmic. As w increases, the AREs of both INT and ARMHH drop first and then stabilize. Specifically, INT gets the minimum ARE when $w > 200$, while ARMHH gets the minimum ARE when $w > 1000$; when w increases from 200 to 1000, the ARE of INT stabilizes, while the ARE of ARMHH drops sharply. $w = 200$ is adequate for INT-CM, while $w = 1000$ is appropriate for the ARMHH-CM under this condition. Moreover, the value of t has a influence on the AREs of both INT and ARMHH, which slightly increases the AREs.

Accordingly, we can allocate suitable memory for the sketch array to a better trade-off between accuracy and memory overhead in ARMHH. Moreover, the results in Figure 9 indicates that $w = 1500$ is adequate in section V-B.

We also record the sliding window memory access from the data plane and the control plane. The data plane accesses the sliding window twice on average and more than twice, with a probability of fewer than 3%, as our trace has a stable flow rate.

VI. Related Work

The sliding window method has some applications in the field of network measurement. Sun et al. [25] use a sliding window-based dynamic timeout strategy to detect and analyze UDP flows. ARMHH can monitor both TCP and UDP flows via sliding window. Ahmed et al. [26] propose a sliding window-based change detection algorithm for asymmetric traffic. ARMHH can be extended to achieve this by comparing the HH set at different epochs. In this paper, we only show HH detection with ARMHH to reveal its advantages in network measurement.

Ding et al. [27] propose an anomaly detection approach based on an isolation forest algorithm for streaming data using the sliding window method, while Ren et al. [28] establish a dynamic Markov model in the sliding window to achieve anomaly detection. These models are accurate and rapid. However, complicated models of this kind are not well supported by the SDN data plane due to the rigid restrictions it implements. By contrast, ARMHH is good at integrating with sketch algorithms. We may test ARMHH with more complicated sketch algorithms in future works.

Unlike the software methods described above, ARMHH can inspect every packet, benefiting from the high throughput of the SDN hardware. We may implement ARMHH on Tofino [29] programmable switches in future.

Network-wide network measurement [8], [11], [30], [31] is important in the large-scale network context. These works achieve network-wide measurement through coordination between different network devices. We may extend ARMHH across multiple devices to achieve network-wide measurement in future.

VII. Conclusion

Our work explores the opportunities offered by replacing interval method in sketch algorithms with sliding window method. We observe that sliding window is more accurate in Heavy Hitter detection, as it eliminates the false negatives between different intervals produced by interval method; moreover, sliding window can report Heavy Hitters in real time.

We proposes ARMHH, a framework targeting *Accurate, Rapid and Memory-efficient Heavy Hitter* detection in the SDN context, which integrates the sliding window approach with sketch algorithms. Notably, ARMHH improves the accuracy by $121.8\times$, $2.8\times$, and $3.5\times$ on average for CM, CU, and Hashpipe, respectively, with additional memory for sliding window. We compressed the sliding window memory by 70%-72% for the queue implement and 62%-68% for the linked list implement.

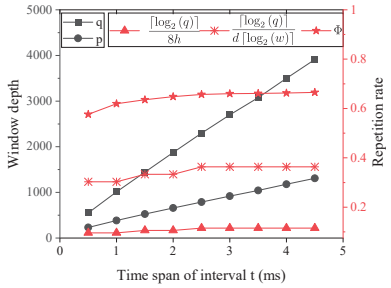


Fig. 7: Parameters in sliding window.

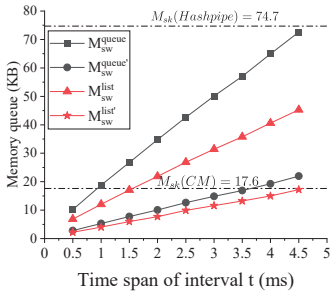


Fig. 8: Memory overhead.

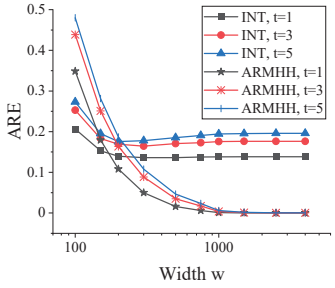


Fig. 9: ARE under different w.

The configuration presented in this research is best for the trace used herein. ARMHH will obtain the best results if our memory allocation method is followed.

REFERENCES

[1] Y. Yuan, S. S. Adhatarao, M. Lin, Y. Yuan, Z. Liu, and X. Fu, "Ada: Adaptive deep log anomaly detector," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 2449–2458.

[2] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4. ACM, 2002, pp. 89–102.

[3] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: High precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 44–58.

[4] Z. Cooper, H. Nguyen, N. Shekita, and F. S. Morton, "Out-of-network billing and negotiated payments for hospital-based physicians: The cost impact of specialists who bill patients at out-of-network rates even though the patients do not choose and cannot avoid these specialists, such as anesthesiologists." *Health Affairs*, vol. 39, no. 1, pp. 24–32, 2020.

[5] Cisco, "Cisco ios netflow," 2020. [Online]. Available: https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html

[6] sFlow.org, "sflow," 2020. [Online]. Available: http://www.sflow.org/

[7] M. Canini, D. Fay, D. J. Miller, A. W. Moore, and R. Bolla, "Per flow packet sampling for high-speed network monitoring," in *2009 First International Communication Systems and Networks and Workshops*. IEEE, 2009, pp. 1–10.

[8] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 334–350.

[9] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 113–126.

[10] R. Jang, D. Min, S. Moon, D. Mohaisen, and D. Nyang, "Sketchflow: Per-flow systematic sampling using sketch saturation event," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1339–1348.

[11] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 576–590.

[12] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 561–575.

[13] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International Conference on Database Theory*. Springer, 2005, pp. 398–412.

[14] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient measurement on programmable switches using probabilistic recirculation," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 313–323.

[15] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k and frequency estimation," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

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

[17] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2002, pp. 323–336.

[18] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Smoking out the heavy-hitter flows with hashpipe," *arXiv preprint arXiv:1611.04825*, 2016.

[19] ONF, "Software-defined networking (sdn) definition," 2019. [Online]. Available: https://www.opennetworking.org/sdn-definition/?nab=1&utm_referrer=https%3A%2F%2Fwww.google.com%2F

[20] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières, "Tiny packet programs for low-latency network control and monitoring," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, 2013, pp. 1–7.

[21] G. Cormode and S. Muthukrishnan, "What's new: Finding significant differences in network data streams," *IEEE/ACM Transactions on Networking*, vol. 13, no. 6, pp. 1219–1232, 2005.

[22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[23] P. language consortium, "Behavioral model version 2," 2019. [Online]. Available: https://github.com/p4lang/behavioral-model

[24] CAIDA, "Caida: Anonymized internet trace 2018," CAIDA, 2019, http://www.caida.org/home/.

[25] Y. Sun, Z. Tian, J. Ye, H. Zhang, and S. Tang, "Udp flow detection and analysis using dynamical timeout strategy based on sliding-window," in *IEEE Conference Anthology*. IEEE, 2013, pp. 1–4.

[26] E. Ahmed, A. Clark, and G. Mohay, "A novel sliding window based change detection algorithm for asymmetric traffic," in *2008 IFIP International Conference on Network and Parallel Computing*. IEEE, 2008, pp. 168–175.

[27] Z. Ding and M. Fei, "An anomaly detection approach based on isolation forest algorithm for streaming data using sliding window," *IFAC Proceedings Volumes*, vol. 46, no. 20, pp. 12–17, 2013.

[28] H. Ren, Z. Ye, and Z. Li, "Anomaly detection based on a dynamic markov model," *Information Sciences*, vol. 411, pp. 52–65, 2017.

[29] Barefoot, "Barefoot tofino: World's fastest p4-programmable ethernet switch asics," 2019. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino/

[30] D. Jiang, Z. Xu, P. Zhang, and T. Zhu, "A transform domain-based anomaly detection approach to network-wide traffic," *Journal of Network and Computer Applications*, vol. 40, pp. 292–306, 2014.

[31] G. Yang, H. Jin, M. Kang, G. J. Moon, and C. Yoo, "Network monitoring for sdn virtual networks," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1261–1270.