HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows

Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen and Xiaoming Li

***Abstract*—Finding top-k elephant flows is a critical task in network traffic measurement, with many applications in congestion control, anomaly detection and traffic engineering. As the line rates keep increasing in today's networks, designing accurate and fast algorithms for online identification of elephant flows becomes more and more challenging. The prior algorithms are seriously limited in achieving accuracy under the constraints of heavy traffic and small on-chip memory in use. We observe that the basic strategies adopted by these algorithms either require significant space overhead to measure the sizes of all flows or incur significant inaccuracy when deciding which flows to keep track of. In this paper, we adopt a new strategy, called *count-with-exponential-decay*, to achieve space-accuracy balance by actively removing small flows through decaying, while minimizing the impact on large flows, so as to achieve high precision in finding top-k elephant flows. Moreover, the proposed algorithm called HeavyKeeper incurs small, constant processing overhead per packet and thus supports high line rates. Experimental results show that HeavyKeeper algorithm achieves 99.99% precision with a small memory size, and reduces the error by around 3 orders of magnitude on average compared to the state-of-the-art.**

***Index Terms*—HeavyKeeper, Top-k, Sketch, Network measurements, Elephant flow**

I. Introduction

A. Background and Motivation

Finding the largest k flows, also referred to as the top-k elephant flows, is a fundamental network management function, where a flow's ID is usually defined as a combination of certain packet header fields, such as source IP address, destination IP address, source port, destination port, and protocol type, and the size of a flow is defined as the number of packets of the flow. Elephant flows contribute a large portion of network traffic. Many management applications can benefit from a function that can find them efficiently, such as congestion control by dynamically scheduling elephant flows [2], network capacity planning [3], anomaly detection [4], and caching of forwarding table entries [5]. Such a function not only is important in networking measurements [6]–[12], but also has applications beyond networking in areas such as data mining [13]–[15], information retrieval [16], databases [17], and security [18].

In real network traffic, it is well known that the distribution of flow sizes (the number of packets in a flow), is highly skewed [19]–[26], *i.e.*, the majority are mouse flows, while the minority are elephant flows. Mouse flows means flows whose frequencies are very small.

Most flows are small while a few flows are very large. The small flows are usually called *mouse flows*, while the large ones are called *elephant flows*.

Finding the top-k elephant flows (or top-k flows for short) in high-speed networks is a challenging task [27]. Extremely high line rates of modern networks make it practically impossible to accurately track the information of all flows. Consequently, approximate methods have been proposed in the literature and gained wide acceptance [21], [28]–[32]. In order to keep up with the line rates, these algorithms are expected to use on-chip memory such as SRAM whose latency is around 1ns [33], [34], in contrast to a latency of around 50ns when off-chip DRAM is used [34]. However, on-chip memory is small. Adding to the challenge, it is highly desirable to keep per-packet processing overhead small and constant, which helps pipelining.

Traditional solutions to finding the top-k flows follow two basic strategies: *count-all* and *admit-all-count-some*. The count-all strategy relies on a sketch (*e.g.*, CM sketch [21]) to measure the sizes of all flows, while using a min-heap to keep track of the top-k flows. For each incoming packet, it records the packet in the sketch and retrieves from the sketch an estimate $\hat{n}_i$ for the size of the flow f_i that the packet belongs to. If $\hat{n}_i$ is larger than the smallest flow size in the min-heap, it replaces the smallest flow in the heap by flow f_i. As a large sketch is needed to count all flows, these solutions are not memory efficient.

The *admit-all-count-some* strategy is adopted by Frequent [35], Lossy Counting [31], Space-Saving [29] and CSS [28]. These algorithms are similar to each other. To save memory, Space-Saving only maintains a data structure called Stream-Summary to count only some flows (m flows). Each new flow will be inserted into the summary, replacing the smallest existing flow. The initial size of the new flow is set as $\hat{n}_{min}+1$, where $\hat{n}_{min}$ is the size of the smallest flow in the summary. By keeping m flows in the summary, the algorithm will report the largest k flows among them, where $m > k$. It assumes every new incoming flow is an elephant, and expels the smallest one in the summary to make room for the new one. But most flows are mouse flows. Such an assumption causes significant error, especially under tight memory (for a limited value of m).

The preliminary version of this paper has been published in USENIX ATC 2018 [1].

Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong and Xiaoming Li are with Department of Computer and Science, Peking University, Beijing, China. Email: yangtongemail@gmail.com.

Steve Uhlig is with Queen Mary University of London, UK. Email: steve@eecs.qmul.ac.uk.

Shigang Chen is with the Department of Computer Information Science Engineering, University of Florida, US. Email: sgchen@ufl.edu.

In recent years, more and more papers have appeared [36]–[39], introducing a lot of new strategies. For example, the Elastic sketch uses votes to decide whether a flow should be recorded or evicted; HeavyGuardian uses the strategy of exponential decay to reduce the estimated frequency with a probability; Cold Filter uses a two-layer filter to prevent mouse flows from entering some data structures (*e.g.*, Space-Saving, the CM sketch); and Counter Tree uses the strategy of two-dimensional counter sharing and derives mathematical formulas to estimate flow sizes.

B. Our Proposed Solution

In this paper, we propose a new algorithm, HeavyKeeper, based on the strategy introduced from [37], called *count-with-exponential-decay*, which keeps all elephant flows while drastically reducing space wasted on mouse flows. HeavyGuardian can handle five different applications, but not including top-k elephant flows detection, while the algorithm we proposed just focuses on finding top-k elephant flows. Unlike *count-all*, our strategy only keeps track of a small number of flows. Unlike *admit-all-count-some*, we do not automatically admit new flows into our data structure and the vast majority of mouse flows will be by-passed. For a small number of mouse flows that do enter our data structure, they will decay away to make room for true elephants. The decay is not uniform for the flows in our data structure. The design of exponential decay is biased against small flows, and it has a smaller impact on larger flows. This design works extremely well with real traffic traces under small memory.

II. Preliminaries

A. Problem Statement

Simply speaking, finding top-k flows refers to finding the largest k flows. Let $\mathcal{P} = \mathbb{P}_1, \mathbb{P}_2, \cdots, \mathbb{P}_N$ be a network stream with N packets. Each packet $\mathbb{P}_l$ $(1 \leqslant l \leqslant N)$ belongs to a flow f_i, where $f_i \in \mathcal{F} = \{f_1, f_2, \cdots, f_M\}$ and $\mathcal{F}$ is the set of flows. Let n_i be the real flow size of flow f_i in $\mathcal{P}$. We order all flows $(f_1, f_2, \cdots, f_M)$ so that $n_1 \geqslant n_2 \geqslant \cdots \geqslant n_M$.

Given an integer k and a network stream $\mathcal{P}$, the output of top-k is a list of k flows from $\mathcal{F}$ with the largest flow sizes, *i.e.*, $f_1, f_2, \cdots, f_k$.

B. Prior Art and Limitations

The count-all strategy: As mentioned above, the *count-all* strategy uses sketches (such as the CM sketch [21] or the Count sketch [30]) to record the sizes of all flows, and uses a min-heap to keep track of the top-k flows, including the flow IDs and their flow sizes. Take the CM sketch as an example. It records packets in a CM sketch, consisting of a pool of counters. For each arrival packet, it hashes the packet's flow ID f to d counters and increases these d counters by one. The smallest value of the d counters is used as the estimated size of the flow, which is used to update the min-heap.

The problem is that all flows are pseudo-randomly mapped to the same pool of counters through hashing. Each counter may be shared by multiple flows, and thus record the sum of sizes of all these flows. Consequently, a small flow may be treated as an elephant flow if all its d counters are shared with real elephant flows.

The admit-all-count-some strategy: As mentioned above, quite a few algorithms use the *admit-all-count-some* strategy, including Frequent [35], Lossy counting [31], and Space-Saving [29].Take Space-Saving as an example. It counts only the sizes of some flows in a data structure called Stream-Summary, which incurs $O(1)$ overhead to search a flow or update the smallest flow. For each arrival packet, if its flow ID is not in the summary, the flow will be admitted into the summary, replacing the smallest existing flow. The new flow's initial size is set to $\hat{n}_{min} + 1$, where $\hat{n}_{min}$ is the smallest flow size in the summary before replacement. A recent work CSS [28] is proposed based on Space-Saving. It inherits the above strategy, but redesigns the data structure of Stream-Summary by using TinyTable [40] to reduce memory usage.

The strategy of *admit-all-count-some* is to admit all new flows while expelling the smallest existing ones from the summary. To give new flows a chance to stay in the summary, their initial flow sizes are set as $\hat{n}_{min} + 1$. Such a strategy drastically over-estimates sizes of flows, and we show an example here. Assume $\hat{n}_{min} = 10,000$ and the summary is already full. Given a new flow, it will directly replace the flow with the smallest size in the summary and set its size to be $10,001$. Therefore, numerous mouse flows will cause significant over-estimation errors.

III. The Design of HeavyKeeper

In this section, we present the data structure and algorithm of our HeavyKeeper, and show how to find the top-k flows accurately and efficiently.

A. Rationale

We aim to use a small hash table to store all elephant flows. As there are a great number of flows, each bucket of the hash table will be mapped by many flows, and we aim to store only the largest flow with its size, which cannot be achieved with no error when using small memory. Therefore, we leverage a probabilistic method called *exponential-weakening decay*. Specifically, when the incoming flow is not found in the hashed bucket, we decay the flow size with a probability, which exponentially decreases as the flow size increases. If the flow size is decayed to 0, it replaces the original flow with the new flow. In this way, mouse flows can easily be decayed to 0, while elephant flows can easily keep stable in the bucket. There are two shortcomings: 1) With a small probability we elect the wrong flow as the largest flow; 2) The reported flow size might be under-estimated because of the decay operations. To address these problems, we use multiple hash tables with different hash functions. An elephant flow could be stored in multiple hash tables, we choose the recorded largest size, minimizing the error of flow sizes.

B. The HeavyKeeper Structure

As shown in Figure III-B, HeavyKeeper is comprised of d arrays, and each array is comprised of w buckets. Each bucket consists of two fields: a fingerprint field and a counter field.[1]

[1] The fingerprint of a flow is a hash value generated by a certain function (for example, if we use $h_f(.)$ as the fingerprint hash function, the fingerprint of flow f_j is $h_f(f_j)$). Although there can be hash collisions among flows, the probability is quite small. For example, if we set the fingerprint size to 16 bits, and there are 10000 buckets in the array, the probability of fingerprint collisions is $1.52 * 10^{-3}$.

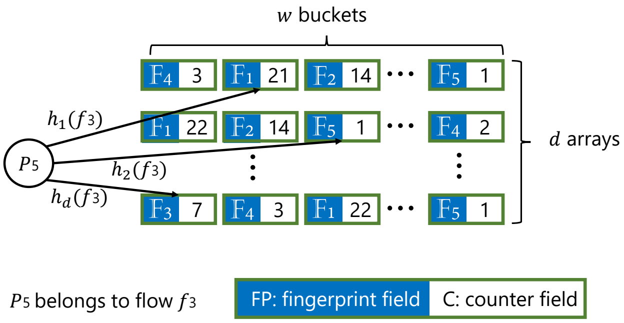


Fig. 1. The data structure of HeavyKeeper.

For convenience, we use $A_j[t]$ to represent the t^{th} bucket in the j^{th} array, and use $A_j[t].FP$ and $A_j[t].C$ to represent its fingerprint field and counter field, respectively. Arrays $A_1...A_d$ are associated with hash functions $h_1(.)...h_d(.)$, respectively. These d hash functions $h_1(.)...h_d(.)$ need to be pairwise independent.

Insertion: Initially, all fingerprint fields are *null*, and all counter fields are 0. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, HeavyKeeper computes the d hash functions, and maps f_i to d buckets $A_j[h_j(f_i)]$ $(1 \leqslant j \leqslant d)$ (one bucket in each array), which we call ***d* *mapped buckets*** for convenience. As shown in Figure III-B, for each mapped bucket, HeavyKeeper applies different strategies for the following three cases:

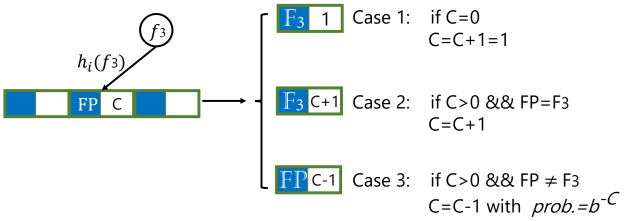


Fig. 2. The main insertion cases of HeavyKeeper. Note: 1) $\mathbb{F}_3$ is the fingerprint of flow f_3. 2) $b > 1$ and $b \approx 1$ (*e.g.*, $b = 1.08$). 3) In Case 3, when C is decayed to 0, the fingerprint field will be replaced by $\mathbb{F}_3$, and then counter C is set to 1.

Case 1: When $A_j[h_j(f_i)].C = 0$. It means that no flow has been mapped to this bucket, then HeavyKeeper sets $A_j[h_j(f_i)].FP = \mathbb{F}_i$ and $A_j[h_j(f_i)].C = 1$, where $\mathbb{F}_i$ represents the fingerprint of f_i.

Case 2: When $A_j[h_j(f_i)].C > 0$ and $A_j[h_j(f_i)].FP = \mathbb{F}_i$. It means $A_j[h_j(f_i)].C$ is possibly the estimated size of f_i. In this case, HeavyKeeper increments $A_j[h_j(f_i)].C$ by 1.

Case 3: When $A_j[h_j(f_i)].C > 0$ and $A_j[h_j(f_i)].FP \neq \mathbb{F}_i$. It means that $A_j[h_j(f_i)].C$ is not the estimated size of f_i. In here, HeavyKeeper applies the *exponential-weakening decay* strategy to this bucket: it decays $A_j[h_j(f_i)].C$ by 1 with a probability P_{decay}. After decay, if $A_j[h_j(f_i)].C = 0$, HeavyKeeper replaces $A_j[h_j(f_i)].FP$ with $\mathbb{F}_i$, and sets $A_j[h_j(f_i)].C$ to 1. Therefore, as long as flows are mapped to a bucket, its counter field will never be 0.

Note that at any time the values of counters are non-negative, since decay only happens in Case 3 and Case 3 happens only when the value of the counter is larger than 0. And in Case 3, when a counter is decayed to zero, the new flow is inserted to this bucket and the counter is set to be 1 immediately.

Query: To query the size of a flow f_i, HeavyKeeper first computes the d hash functions to get d buckets $A_j[h_j(f_i)]$ $(1 \leqslant j \leqslant d)$. Among the d mapped buckets, it chooses those buckets whose fingerprint fields are equal to $\mathbb{F}_i$. It then reports the maximum counter field of those buckets, *i.e.*, $max_{1 \leqslant j \leqslant d}\{A_j[h_j(f_i)].C\}$ where $A_j[h_j(f_i)].FP = \mathbb{F}_i$.

For convenience, for those d mapped buckets of f_i, if $A_j[h_j(f_i)].FP = \mathbb{F}_i$, we say that f_i is ***held*** at bucket $A_j[h_j(f_i)]$. Ignoring the limited impact of fingerprint collisions, we prove that *the reported size for each flow is equal to or smaller than the real flow size* in Section A. If a flow is *held* at no mapped bucket, it reports that it is a mouse flow. If a flow is *held* at multiple buckets, HeavyKeeper reports the maximum counter field.

Decay probability: The key problem is how to choose the function to calculate the probability. Based on our experiments in real and synthetic datasets, we find that as long as the parameters are set reasonably, functions satisfying the following condition all have good performance: the larger the value in the current counter field is, the smaller the probability is. We finally choose the exponential function

$$P_{decay} = b^{-C} \qquad (b > 1)$$

where C is the value in the current counter field, and b ($b > 1$ and $b \approx 1$, *e.g.*, $b = 1.08$) is a predefined exponential base number. It is because this function has the following properties in addition. 1) As the value increases, the rate of probability reduction gradually increases. 2) When the value is large enough (*e.g.*, 50), the probability is close to 0, so we can regard the probability as 0, accelerating the throughput of our algorithm. 3) When the value is small (*e.g.*, 3), the recorded flow can hardly be an elephant flow, and at the same time the probability is close to 1, which exactly matches this condition. On the contrary, most other kinds of functions (*e.g.*, linear functions, polynomial functions) do not have all the above properties.

Therefore, the larger size a flow has, the harder its size is decayed. For elephant flows, it is held at several buckets, and the corresponding counter fields are incremented regularly, while decayed with a very small probability. Therefore, the error rate for estimated sizes of elephant flows is very small.

Note: Our data structure of d arrays and d 2-way independent hash functions may show some similarity with that of CM [21]. But similarity stops there. CM records the sizes of all flows; we record the sizes of a small number of flows. CM does not store flow IDs; we do. CM stores information of each flow in d counters; we keep each flow mostly in one bucket, while d-hashing helps find an empty bucket. CM does not have to worry about the issue of kicking out existing flows to make room for new ones, which is what our exponential delay does.

Example: As shown in Figure III-B, given an incoming packet $\mathbb{P}_5$ belonging to flow f_3, we compute the d hash functions to obtain one bucket in each array. In the mapped bucket of the first array, the fingerprint field is not equal to $\mathbb{F}_3$ and the counter field is 21, thus we decay the counter field from 21 to 20 with a probability of 1.08^{-21} (assume $b = 1.08$). In the second mapped bucket, the fingerprint field is not $\mathbb{F}_3$ yet, and with a probability of 1.08^{-1}, we decay the counter field from 1 to 0. If the counter field is decayed to 0, we set the fingerprint field to $\mathbb{F}_3$, and set the counter field to 1. In the last mapped bucket, the fingerprint field is $\mathbb{F}_3$, we increment the counter field from 7 to 8.

Analysis: HeavyKeeper uses fingerprint to identify and keep elephant flows. If a mouse flow with a small flow size is held at a bucket, it will be replaced by other flows mapped to this bucket soon, because each flow mapped to this bucket with a different fingerprint will decay the counter field with a high probability ($b^{-C} \to 1$ when C is small). If an elephant flow is held at a bucket, the corresponding counter field can easily be incremented to a large value since elephant flows have many incoming packets. Moreover, the decay probability becomes very small ($b^{-C} \to 0$ when C is large) as the counter field increases to a large value. Therefore, mouse flows can hardly be held in HeavyKeeper for a long time, and thus have a large probability to be *passers-by* of HeavyKeeper. However, elephant flows can keep stable in HeavyKeeper, and the estimated sizes of elephant flows are accurate.

C. Basic Version for Finding Top-k Elephant Flows

To find top-k elephant flows, our basic version just uses a HeavyKeeper and a min-heap. The min-heap is used to store the IDs and sizes of top-k flows. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, we first insert it into HeavyKeeper. Suppose that HeavyKeeper reports the size of f_i as $\hat{n}_i$. If f_i is already in the min-heap, we update its estimated flow size with $max(\hat{n}_i, min\_heap[f_i])$, where $min\_heap[f_i]$ is the recorded size of f_i in min-heap. Otherwise, if $\hat{n}_i$ is larger than the smallest flow size which is in the root node of the min-heap, we expel the root node from the min-heap, and insert f_i with $\hat{n}_i$ into the min-heap. To query top-k flows, we simply report the k flows in the min-heap with their estimated flow sizes.

Note that in our implementation, *we use Stream-Summary instead of min-heap*, as the function of min-heap and Stream-Summary is similar, and Stream-Summary can achieve O(1) update complexity. For better understanding, we use min-heap to explain in our paper.

D. Optimizations

In this section, we propose further optimization methods to avoid accidental errors and improve speed. For convenience, we use n_{min} to denote the minimal flow size in the min-heap.

Optimization I: Fingerprint Collisions Detection.

Problem: Assume that there is a bucket in HeavyKeeper where flow f_i is held, and a mouse flow f_j mapped to the same bucket has the same fingerprint as f_i, *i.e.*, $\mathbb{F}_i = \mathbb{F}_j$ due to hash collisions. Then, the mouse flow f_j is also held at this bucket, and its estimated size is drastically over-estimated. In the worst case, if flow f_j has a fingerprint collision in all d arrays, the mouse flow f_j will probably be inserted into the min-heap. It can hardly be expelled due to its drastically over-estimated size. One may argue that we can store the entire IDs of flows instead of using fingerprints, which can definitely avoid hash collisions. However, in real data streams, the number of bits of a flow's ID is usually very large (*e.g.*. more than 100 bits in 5-tuple), leading to a memory waste. Indeed, the better the memory efficiency, the higher the accuracy of algorithms. So the goal of our design is to find a solution to alleviate hash collisions without increasing the number of recorded bits. Therefore, we propose a solution based on the following Theorem.

Theorem 1. *When there is no fingerprint collision, after a flow f_i is inserted into HeavyKeeper, if its estimated size $\hat{n}_i$ is larger than n_{min} (recall that we use n_{min} to denote the minimal flow size in the min-heap), then we must have*

$$\hat{n}_i = n_{min} + 1$$

The proof of this Theorem is not hard to derive and we skip it due to space limitations.

Solution: Based on Theorem 1, if f_i is not in the min-heap but $\hat{n}_i > n_{min} + 1$, then f_i is a mouse flow whose size is drastically over-estimated due to fingerprint collision. Therefore, we should not insert f_i into the min-heap in this case. Each time when we want to insert a flow into the min-heap, we need to check as above. For example, when a mouse flow e arrives, it might increase a counter's value due to fingerprint collision. In this way, the estimated frequency of e will be very large, and it might be inserted into the min-heap by mistake. However, due to the strategy of our fingerprint collision detection, before inserting flow e into the min-heap, we check whether the estimated frequency of e is larger than $n_{min}+1$. In other words, e will be inserted into the min-heap only when its estimated frequency is equal to $n_{min}+1$, whose probability is very low.

Optimization II: Selective Increment.

Problem: If a flow f_i is not in the min-heap, then the estimated flow size should be no larger than n_{min}. However, due to fingerprint collisions, there could be some mapped buckets of flow f_i where the fingerprint field is $\mathbb{F}_i$ and the counter field is larger than n_{min}. In this case, flow f_i is not the flow that is held at this bucket, and thus increasing the corresponding counter field can only incur extra error.

Solution: In this case, instead of incrementing or decaying the corresponding counter field, we make no change.

E. Hardware Parallel Version

Based on the basic version, we propose a new version using the above two optimization methods. It is called `Hardware Parallel version` (Parallel version for short) because for each insertion, the operation in each array can be implemented in parallel on hardware platforms (*e.g.*, FPGA, ASIC, or P4Switch). We will propose a more accurate version (named `Software Minimum version`, Minimum version for short in Section IV) at the cost of sacrificing the parallel property. The insertion and query processes of the Parallel version of our algorithm are presented as follows (see pseudo-code in Algorithm 1).

Insertion: All counters and fingerprints in HeavyKeeper and the min-heap are initialized to 0. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, these are the following three steps for each insertion:

Step 1: We check whether flow f_i is already monitored by the min-heap, which is shown in line 1-3 in Algorithm 1. For convenience, we use a boolean variable $flag$ to represent the result.

Step 2: We insert f_i into HeavyKeeper, which is shown in line 4-22 in Algorithm 1. According to Optimization II, for each mapped bucket, if the fingerprint field is equal to $\mathbb{F}_i$, we increment the counter field only when $flag = true$ or $C < n_{min}$, where C is the original value in the counter field.
Step 3: We get an estimated size $\hat{n}_i$ of flow f_i from HeavyKeeper, which is shown in line 23-27 in Algorithm 1. According to Optimization I, if $flag$ is $true$, we update the estimated size of flow f_i in the min-heap with $\hat{n}_i$. If $flag$ is $false$, we insert flow f_i into the min-heap with $\hat{n}_i$ in only two cases: 1) the number of flows that are in the min-heap is less than k; 2) $\hat{n}_i = n_{min} + 1$.

Algorithm 1: Insertion process for finding top-k flows.

```
   Input: A packet $\mathbb{P}_l$ belonging to flow $f_i$
1  $flag \leftarrow false$;
2  if $f_i \in min\_heap$ then
3      $flag \leftarrow true$;
4  $HeavyK\_V \leftarrow 0$;
5  for $j \leftarrow 1$ to $d$ do
6      $C \leftarrow A_j[h_j(f_i)].C$;
7      if $C = 0$ then
8          $A_j[h_j(f_i)].FP \leftarrow \mathbb{F}_i$;
9          $A_j[h_j(f_i)].C \leftarrow 1$;
10         $HeavyK\_V \leftarrow max(HeavyK\_V, 1)$;
11     else if $A_j[h_j(f_i)].FP = \mathbb{F}_i$ then
12         if $flag = true$ or $C < min\_heap.n_{min}$ then
13             $A_j[h_j(f_i)].C++$;
14         $HeavyK\_V \leftarrow$
             $max(HeavyK\_V, A_j[h_j(f_i)].C)$;
15     else if $rand() < b^{-C}$ then
16         $A_j[h_j(f_i)].C--$;
17         if $A_j[h_j(f_i)].C = 0$ then
18             $A_j[h_j(f_i)].FP \leftarrow \mathbb{F}_i$;
19             $A_j[h_j(f_i)].C \leftarrow 1$;
20             $HeavyK\_V \leftarrow max(HeavyK\_V, 1)$;
21 if $flag = true$ then
22     $min\_heap[f_i] \leftarrow max(HeavyK\_V, min\_heap[f_i])$;
23 else
24     if $min\_heap$ has empty buckets or
         $HeavyK\_V - n_{min} = 1$ then
25         $min\_heap.insert(f_i)$;
```

Query top-k flows: It reports the k flows recorded in the min-heap and their estimated flow sizes.
Analysis: Since HeavyKeeper achieves very small error rate on the flow size estimation of elephant flows, it can significantly reduce the error in finding top-k elephant flows. Furthermore, the first two optimizations reduce the impact of fingerprint collisions, and enhance the precision of finding top-k elephant flows and their flow size estimation.

F. Limitation

As mentioned before, when the *exponential-weakening decay* is performed on a bucket, if its counter value is large enough (*e.g.*, 50), the probability of reducing its value is close to 0. Therefore, in the worst case, when a new flow arrives, if all values of its mapped d counters are large enough, it could never be inserted into some buckets. In fact, this limitation means that the current memory size is too tight to record top-k elephant flows. To avoid this situation, we can use an extra counter to record how many times this situation happens. As long as the value of the extra counter is larger than a predefined threshold, we add a new array, *i.e.*, the $d+1^{th}$ array. In this way, the new flow will have a chance to record its information.

Besides, our proposed algorithm cannot handle other flow measurement tasks (*e.g.*, flow size estimation, entropy detection) and cannot support weighted updates. However, thanks to the fact that HeavyKeeper is designed to handle top-k flows detection, it achieves higher accuracy than other related algorithms, which will be detailed in Section VI-E.

G. Other uses of HeavyKeeper

Besides finding top-k flows in a network stream, HeavyKeeper can also perform other tasks in network traffic measurement, such as heavy hitter detection and change detection. Due to space limitations, here we only briefly introduce how to perform these tasks using HeavyKeeper.
Heavy hitter detection: Given a threshold θ, a heavy hitter [20] is a flow whose size $n_i > \theta N$, where N is the number of packets in total. The heavy hitter detection algorithm is very similar to that of finding top-k flows. The only difference is that when querying heavy hitters, it reports those flows whose estimated size is larger than θN in min-heap.
Change detection: The network stream is divided into fixed-size time bins. Given a flow, if the difference of its flow sizes in two adjacent time bins is larger than a predefined threshold, then the flow is called a heavy change [20], [41]. We use the very flow ID as the fingerprint of each flow. For two adjacent time bins, we insert their packets into two different HeavyKeepers. By comparing buckets in the corresponding location in the two HeavyKeepers, we obtain the estimated difference of sizes of the flows, and report the heavy changes by checking if the difference is larger than the threshold.

IV. SOFTWARE MINIMUM VERSION

In the above section, we describe the Hardware Parallel Version of HeavyKeeper, in which all the d arrays can be inserted or queried in parallel. We observe that its accuracy can be further improved by sacrificing the parallel property. In this section, we propose the Software Minimum Version to further enhance the accuracy.

A. Problem

We observe that it is unnecessary to decay all the mapped counters in the basic version. Specifically, when inserting an incoming packet $\mathbb{P}_l$ belonging to flow f_i, HeavyKeeper computes d hash functions and maps f_i to d buckets $A_j[h_j(f_i)]$ $(1 \leqslant j \leqslant d)$ (one bucket in each array). For each bucket, HeavyKeeper applies different strategies for three different cases. We focus on the third case below. In **Case 3**, $A_j[h_j(f_i)].C > 0$ and $A_j[h_j(f_i)].FP \neq \mathbb{F}_i$, HeavyKeeper decays $A_j[h_j(f_i)].C$ by 1 with a probability P_{decay}, and after decay, if $A_j[h_j(f_i)].C = 0$, HeavyKeeper replaces

$A_j[h_j(f_i)].FP$ with $\mathbb{F}_i$, and sets $A_j[h_j(f_i)].C$ to 1. However, for a bucket $A_k[h_k(f_i)]$ $(1 \leqslant k \leqslant d)$ in HeavyKeeper where an elephant flow f_i is held, if another flow f_j is mapped to the same bucket due to hash collisions, *i.e.*, $f_i \neq f_j$ *and* $\mathbb{F}_i = \mathbb{F}_j$, then $A_k[h_k(f_i)]$ is decayed by 1 with a probability P_{decay}, but such decay is not always necessary and could be harmful for the following reasons.

First, if f_j is a mouse flow which only has a few packets, the elephant flow f_i can hardly be replaced by it, but f_i's counter field is possibly decayed for a few times (*e.g.*, decayed from 1000 to 999). Such decay can hardly cause a replacement, but at the same time, it makes f_i's recorded flow size in this bucket less than its real flow size, which will degrade the accuracy of queries. One may argue that f_i will always occupy a bucket if we do not perform any decay on it. In the worst case, if f_i is not an elephant flow, this strategy will make new flows do not have a choice to be inserted into that bucket. However, this situation happens only when for a new flow, all values of its mapped d counters are very large. As mentioned in Section IV-C, we can use an extra counter and automatically add a new array to avoid this situation.

Second, if f_j is an elephant flow which has a large number of packets, whether $A_k[h_k(f_i)].C$ will be decayed to 0 and $A_k[h_k(f_i)].FP$ will be replaced with $\mathbb{F}_j$ depends on the following packets of f_i and f_j. In such a contest of the two elephant flows, the counter in this bucket will be decayed many times. There are two results. 1) If f_i wins and keeps held in this bucket, *i.e.*, $A_k[h_k(f_i)].C$ never reaches 0, $A_k[h_k(f_i)].C$ will be much less than the real flow size of f_i. When querying the size of flow f_i, HeavyKeeper reports the maximum counter field of all the mapped buckets. As an elephant flow, f_i is likely to be kept in several buckets, and the counter fields in other buckets may well be larger than $A_k[h_k(f_i)].C$, so $A_k[h_k(f_i)].C$ makes no contribution to the accuracy of queries. 2) If f_j wins and replaces f_i in the bucket $A_k[h_k(f_i)]$, $A_k[h_k(f_i)].C$ is much less than the real flow size of f_j. Also, this counter makes no contribution to the query results of flow f_j.

In summary, *it is unnecessary and unhelpful to decay large counters.*

B. Solution: Minimum Decay

To address the above problem, we propose a solution, and the key technique is called "Minimum Decay". Its key idea is that we choose to decay the smallest one instead of decaying all the mapped counters. Below we show the details of our solution. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, HeavyKeeper computes d hash functions and maps f_i to d buckets $A_j[h_j(f_i)]$ $(1 \leqslant j \leqslant d)$ (one bucket in each array). For the d mapped buckets, suppose $\mathbb{F}_i$ is the fingerprint of f_i. There are three situations.

Situation 1: If one of the d mapped buckets has the same fingerprint as $\mathbb{F}_i$, we just increment the corresponding counter by 1.

Situation 2: If all d mapped buckets do not have the fingerprint $\mathbb{F}_i$, but one or more of the mapped buckets are empty. In this situation, we just insert f_i into the first empty bucket.

Situation 3: If all d mapped buckets are full and do not have the fingerprint $\mathbb{F}_i$. In this situation, we choose the smallest counter among the mapped bucket, and then perform the decay operation. If there is more than one smallest counter, we only choose the first one to decay.

Note that for each insertion, we only update one mapped bucket, and do nothing for other mapped buckets.

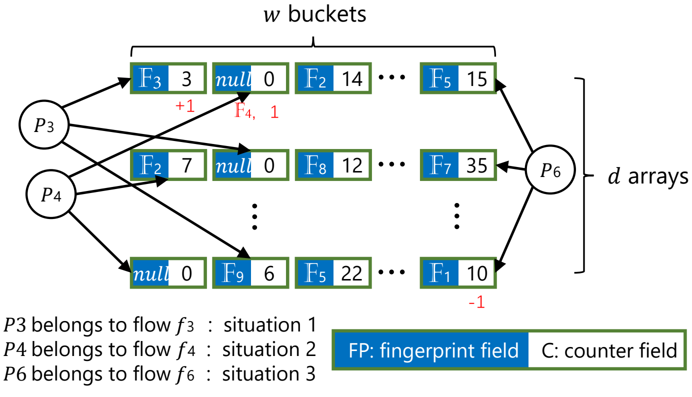


Fig. 3. Examples of the insertion of Parallel version.

Examples: Figure IV-B shows three incoming packets corresponding to the three situations, respectively. Given each incoming packet, we compute the d hash functions to obtain one bucket in each array. We only show the first, second and last array for convenience. For packet $\mathbb{P}_3$ belonging to flow f_3, the first mapped bucket holds the same fingerprint as f_3 ($\mathbb{F}_3$), so this is the above *Situation 1*. Thus we increment the counter field from 3 to 4. For packet $\mathbb{P}_4$ belonging to flow f_4, none of the d mapped buckets holds the fingerprint $\mathbb{F}_4$, but there are two empty buckets, so this is *Situation 2*. We insert flow f_4 into the mapped bucket in the first array. We set its fingerprint field to $\mathbb{F}_4$ and its counter field to 1. For packet $\mathbb{P}_6$ belonging to flow f_6, none of the d mapped buckets holds the fingerprint $\mathbb{F}_6$ and none of them is full, so this is *Situation 3*. The counter field in the last mapped bucket is the smallest, so we decay it by 1 with a probability of 1.08^{-10}, and do nothing to the other mapped buckets.

C. Hardware Minimum Version for Finding Top-k Flow

Based on the Hardware Parallel version, we propose the Software Minimum version (Minimum version for short) using the above minimum decay technique. The insertion and query processes of our Minimum version of our algorithm are presented as follows. Due to space limitation, we present the pseudo-code in the Appendix of our technical report [42].

Insertion: All counters and fingerprints in HeavyKeeper and the min-heap are initialized to 0. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, there are the following five steps for each insertion:

Step 1: We check whether flow f_i is already monitored by the min-heap, denoted by a bloolean variable $flag$.

Step 2: We check whether there is a mapped bucket holding the same fingerprint as $\mathbb{F}_i$. If there is and the corresponding bucket could be updated ($flag = true$ or the value of counter is less than n_{min}), we increment the corresponding counter filed by 1, and then go to step 5; otherwise, we go to step 3.

Step 3: We check whether there is a mapped bucket that is empty. If there is, we insert this packet into the first empty bucket and then go to step 5; otherwise, we go to step 4.

Step 4: We choose the bucket with the smallest counter field among the d mapped buckets and decay it with a certain probability. If there is more than one such bucket, we only decay the first one.

Step 5: Step 5 is similar to step 3 of Parallel version of HeavyKeeper. We get an estimated size $\hat{n}_i$ of flow f_i from HeavyKeeper. If $flag$ is $true$, we update the estimated size of flow f_i in the min-heap with $\hat{n}_i$. If $flag$ is $false$, we insert flow f_i into the min-heap with $\hat{n}_i$ in only two cases: 1) the number of flows that are in the min-heap is less than k; 2) $\hat{n}_i = n_{min} + 1$.

Query top-k flows: We report the k flows recorded in the min-heap and their estimated flow sizes.

Analysis: The Parallel version of HeavyKeeper achieves fast processing speed and small error rate in finding top-k elephant flows. Based on the Parallel version, the Minimum version further improves the accuracy. Specifically, when inserting a packet, the Minimum version only needs to change at most one bucket, thus it avoids unnecessary and unhelpful decay. Our experimental results (see Figure 23, 26 and 29) verify that the accuracy is significantly improved when using the Minimum Decay technique.

V. Mathematical Analysis

In this section, we first claim that there is no over-estimation of HeavyKeeper, and then derive the formula of error bound in the Minimum version of HeavyKeeper. Note that we also derived the formula of error bound in the basic version of . Due to space limitation, we provide the derivation process of the basic version in the Appendix of our technical report [42].

A. Claim of No Over-estimation Error of HeavyKeeper

Theorem 2. *In the Minimum version, let $n_i(t)$ be the real size of flow f_i at time t, and let $A_j[h_j(f_i)](t).C$ be the counter field of the mapped bucket of flow f_i in the j^{th} array at time t. If there is no fingerprint collision, then*

$$\forall j, t, \;\; A_j[h_j(f_i)](t).C \leqslant n_i(t)$$

Proof. It is not hard to prove this theorem. Due to space limitation, we provide the proof in the Appendix of our technical report [42]. □

B. Error Bound of the Minimum Version of HeavyKeeper

Theorem 3. *Assume that there is no fingerprint collision and once the fingerprint of an elephant flow is inserted into its mapped bucket, it is held there all the time. For any $\epsilon > 0$, assume an elephant flow f_i with size n_i is held in the bucket, we have*

$$Pr\{n_i - \hat{n}_i \geqslant \lceil \epsilon N \rceil\} \leqslant \frac{\gamma}{\epsilon w n_i (b-1)} \tag{1}$$

where w is the width of each array, b the exponential base, and γ the proportion of mouse flows in all flows.

Proof. For convenience, we use N to denote the total number of packets, M to denote the number of different flows and d to denote the number of arrays. Let's focus on the j^{th} array. Flow f_i is correctly reported, so at the end, the fingerprint of flow f_i is held in the $h_j(f_i)^{th}$ bucket of the j^{th} array. Let $I_{i,j,i'}$ be a binary random variable, defined as

$$I_{i,j,i'} = \begin{cases} 0 & (f_i = f_{i'}) \vee (h_j(f_i) \neq h_j(f_{i'})) \\ 1 & (f_i \neq f_{i'}) \wedge (h_j(f_i) = h_j(f_{i'})) \end{cases} \tag{2}$$

$I_{i,j,i'} = 1 \; iff$ different flows f_i and $f_{i'}$ are held at the same bucket in the j^{th} array. We use the three situations the same as Section IV-B. We define binary random variable $Y_i (1 \leqslant i \leqslant M)$ as:

$$Y_i = \begin{cases} 0 & \exists 1 \leqslant j \leqslant d, s.t. \; \forall 1 \leqslant k \leqslant M, I_{i,j,k} = 0 \\ 1 & else \end{cases} \tag{3}$$

As mentioned in Subsection III-B, d hash functions $h_1(.)...h_d(.)$ are pairwise independent, and the following proof is based on this condition.

For each flow f_i, if in the d mapped buckets, there is at least one bucket with no hash collision, $Y_i = 0$. Otherwise, in each of these d mapped buckets, $\exists$ a flow $f_j (f_i \neq f_j)$ that is also mapped to this bucket, then $Y_i = 1$. So if $Y_i = 0$, for any incoming packet $\mathbb{P}$ belonging to f_i, *Situation 3* can never happen. Now let's calculate $E(Y_i)$, the probability that in each of the d arrays, there are hash collisions in the bucket to which flow f_i is mapped. In a given bucket, the probability that a flow is mapped here is $\frac{1}{w}$, so in a bucket to which f_i is mapped, the probability that no other flow is mapped here is $(1 - \frac{1}{w})^{M-1}$. And in a given array, the probability that hash collision happens in the bucket to which f_i is mapped is $(1 - (1 - \frac{1}{w})^{M-1})$, thus,

$$E(Y_i) = \left[1 - (1 - \frac{1}{w})^{M-1} \right]^d \tag{4}$$

We define random variable $X_{i,j}$ as:

$$X_{i,j} = \sum_{i'=1}^{M} I_{i,j,i'} n_{i'} Y_i \tag{5}$$

Among the flows held in the same bucket as flow f_i, except for flow f_i itself, some flows are unlikely to cause *Situation 3*, thus unlikely to decay the counter field of this bucket, and others are likely to. $X_{i,j}$ represents the sum of the sizes of the latter kind of flows.

For each incoming packet, if it belongs to flow f_i, the counter field is incremented by 1; if not, the counter field is not changed or decayed. Thus we have

$$n_i - X_{i,j} \leqslant A_j[h_j(f_i)].C \leqslant n_i \tag{6}$$

Note that $A_j[h_j(f_i)].C$ is the counter value at the query time. Specifically, if for all packets that do not belong to flow f_i, *Situation 3* happens, and when they are being processed, this counter field is the smallest one in all d mapped buckets, and they all decay the counter field, then $A_j[h_j(f_i)].C = n_i - X_{i,j}$. If all such packets do not decay the counter field, then $A_j[h_j(f_i)].C = n_i$. Then we define random variable $P_{i,j,l}$ as the probability that the l^{th} packet decays the counter field, therefore,

$$A_j[h_j(f_i)].C = n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l} \tag{7}$$

For any $\epsilon > 0$, we have the following formula based on the Markov inequality.

$$\begin{aligned} & Pr\{A_j[h_j(f_i)].C \leqslant n_i - \epsilon N\} \\ &= Pr\{n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l} \leqslant n_i - \epsilon N\} \\ &= Pr\{\sum_{l=1}^{X_{i,j}} P_{i,j,l} \geqslant \epsilon N\} \leqslant \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\epsilon N} \end{aligned} \tag{8}$$

Now let's focus on $E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$. Assume that all packets are uniformly distributed. Since we assume that the fingerprint of an elephant flow is held at its mapped bucket since inserted, if the l^{th} packet belongs to an elephant flow, *Situation 3* cannot happen at this moment. That is, if the l^{th} packet is to decay the given counter field, it must be a mouse flow and this counter field is the smallest in all d mapped buckets' counter fields. Recall that $A_j[h_j(f_i)].C$ is the counter value at the query time. We assume that before the query time, when a flow arrives, the counter value is uniformly distributed within the range $[1, A_j[h_j(f_i)].C]$, so the probability that the counter size is equal to any integer within this range is $1/A_j[h_j(f_i)].C$. In addition, the decay happens on condition that 1) the new flow is a mouse flow, whose probability is γ; 2) *Situation 3* happens and 3) this counter is the first smallest counter. The probability of 2) and 3) is no larger than 1. For any C which satisfies $1 \leqslant C \leqslant n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$, we have the following formula:

$$\begin{aligned} Pr\{P_{i,j,l} = \frac{1}{b^{C}}\} &\leqslant \frac{\gamma}{A_j[h_j(f_i)].C} \\ &= \frac{\gamma}{n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})} \end{aligned} \tag{9}$$

Let β be $n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$. As a result,

$$\begin{aligned} E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) &= \sum_{l=1}^{E(X_{i,j})} E(P_{i,j,l}) \\ &\leqslant E(X_{i,j}) \sum_{\mathrm{C}=1}^{\beta} \frac{\gamma}{b^{\mathrm{C}}} \frac{1}{\beta} = \frac{\gamma E(X_{i,j})}{\beta} \cdot \sum_{\mathrm{C}=1}^{\beta} \frac{1}{b^{\mathrm{C}}} \\ &= \frac{\gamma E(X_{i,j})}{\beta} \cdot \frac{\frac{1}{b}\left[1 - (\frac{1}{b})^{\beta}\right]}{1 - \frac{1}{b}} \\ &\leqslant \frac{\gamma E(X_{i,j})\left[1 - (\frac{1}{b})^{n_i}\right]}{n_i(b-1)} \end{aligned} \tag{10}$$

Furthermore, for $E(X_{i,j})$, based on Equation 4 and 5,

$$\begin{aligned} E(X_{i,j}) &= E\left(\sum_{i'=1}^{M} I_{i,j,i'} n_{i'} Y_i\right) \\ &\leqslant \sum_{i'=1}^{M} n_{i'} E(I_{i,j,i'}) E(Y_i) \\ &= \frac{N}{w}\left[1 - \left(1 - \frac{1}{w}\right)^{M-1}\right]^d \end{aligned} \tag{11}$$

Therefore, based on Equation 10,

$$\begin{aligned} E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) &\leqslant \frac{\gamma N\left[1 - (\frac{1}{b})^{n_i}\right]}{w n_i(b-1)}\left[1 - \left(1 - \frac{1}{w}\right)^{M-1}\right]^d \\ &\leqslant \frac{\gamma N}{w n_i(b-1)}\left[1 - \left(1 - \frac{1}{w}\right)^{M-1}\right]^d \end{aligned} \tag{12}$$

Then, based on Equation 8,

$$\begin{aligned} & Pr\{A_j[h_j(f_i)].C \leqslant n_i - \epsilon N\} \\ &\leqslant \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\epsilon N} \\ &\leqslant \frac{\gamma N}{\epsilon N w n_i(b-1)}\left[1 - \left(1 - \frac{1}{w}\right)^{M-1}\right]^d \\ &= \frac{\gamma}{\epsilon w n_i(b-1)}\left[1 - \left(1 - \frac{1}{w}\right)^{M-1}\right]^d \end{aligned} \tag{13}$$

Note that for an elephant flow f_i, n_i is very large, so we have

$$\begin{aligned} Pr\{n_i - \hat{n}_i \geqslant \lceil \epsilon N \rceil\} &\leqslant Pr\{\hat{n}_i \leqslant n_i - \epsilon N\} \\ &\leqslant \frac{\gamma}{\epsilon w n_i(b-1)}\left[1 - \left(1 - \frac{1}{w}\right)^{M-1}\right]^d \\ &\leqslant \frac{\gamma}{\epsilon w n_i(b-1)}\left(1 - e^{\frac{1-M}{w}}\right)^d \end{aligned}$$

Since w and d are much smaller than M, we have $1 - \delta < (1 - e^{\frac{1-M}{w}})^d < 1$, where δ is a very small positive number. Therefore, we have

$$Pr\{n_i - \hat{n}_i \geqslant \lceil \epsilon N \rceil\} \leqslant \frac{\gamma}{\epsilon w n_i(b-1)}$$

Theorem holds. □

Theorem 3 is based on an assumption that for an elephant flow, since it is inserted into a bucket, it would be held there all the time. However, if an elephant flow with extremely large size, say 10^{20}, arrives so late that all of its mapped buckets have been filled with other elephant flows with size 1000, it seems impossible that this flow can be recorded accurately. In order to deal with this limitation that elephant flows arriving late are at a disadvantage, we can use the method mentioned in Section III-F. Specifically, we can use an extra counter to record how many times a flow's d mapped counters are all large counters. If this extra counter value exceeds the predefined threshold, we add a new array to make room for the new flow.

VI. Experimental Results

A. Experiment Setup

Platform: Our experiments are run on a server with dual 6-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62 GB total system memory. Each core has two L1 caches with 32KB memory (one instruction cache and one data cache) and one 256KB L2 cache. All cores share one 15MB L3 cache.

Dataset:

1) Campus dataset: The first dataset is comprised of IP packets captured from the network of our campus. We rely on the usual definition of a flow, through its 5-tuple, *i.e.*, source IP

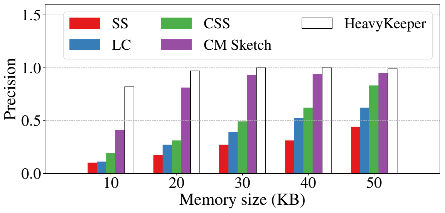


Fig. 4. Precision vs. memory size (Campus).

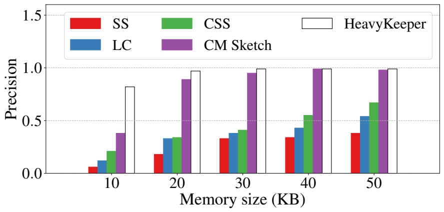


Fig. 5. Precision vs. memory size (CAIDA).

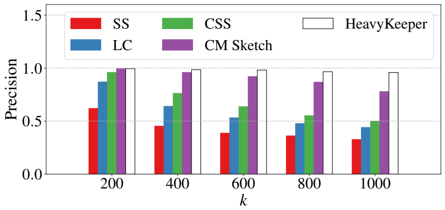


Fig. 6. Precision vs. k (Campus).

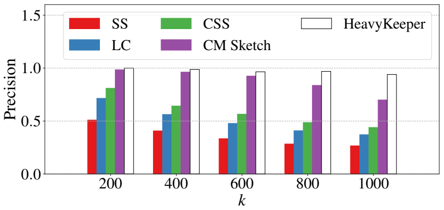


Fig. 7. Precision vs. k (CAIDA).

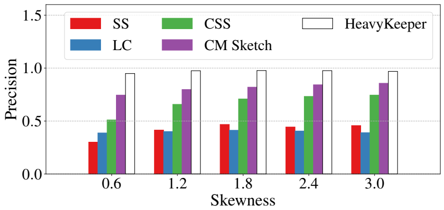


Fig. 8. Precision vs. skewness (Synthetic).

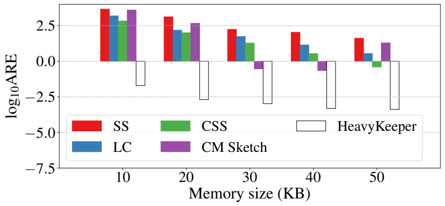


Fig. 9. ARE vs. memory size (Campus).

address, destination IP address, source port, destination port, and protocol type. There are 10M packets in total, belonging to 1M flows. For convenience, we use *campus dataset* to denote this dataset.

2) CAIDA dataset: The second dataset is a CAIDA Anonymized Internet Trace from 2016 [43], made of anonymized IP packets. Each flow in this dataset is identified by the source and destination IP address. We use the first 10M. [2] packets, belonging to about 4.2M flows.

3) Synthetic datasets: We generate 10 different synthetic datasets according to a Zipf [44] distribution with different skewness (from 0.6 to 3.0)[3] Each dataset is comprised of 32M packets, belonging to 1 $\sim$ 10M flows depending on the skewness. Each packet is 4 bytes long. The code of the dataset generator is the one from Web Polygraph [45].

The reason why we use different header fields is to show HeavyKeeper's universality in terms of header fields, *i.e.*, HeavyKeeper can be deployed on different header fields. And the experimental results show that the performances of HeavyKeeper on datasets with different header fields are similar.

Implementation: The implementation of two versions of HeavyKeeper is done in C++. We also implemented in C++ the other related algorithms including Space-Saving (SS), Lossy counting (LC), and CM sketch. The source code of CSS was provided by its author [28], and is written in Java. It is much slower than Space-Saving written in C++. Therefore, we do not include CSS in our speed experiments. For Space-Saving, Lossy counting, and CSS, the **number of buckets** m is determined by the memory size, which is usually much larger than k. When querying top-k flows, they report the largest k flows of them. For CM sketch, the size of the heap is k, the number of arrays is 3, and the width of each array is determined by the memory size. In our algorithm, the number of buckets m in Stream-Summary is equal to k, and HeavyKeeper occupies the rest memory size. Here we set $d = 2$, and w depends on the memory size. Both the fingerprint field and the counter field are 16-bit long. The reason why we just set 16 bits for the fingerprint field is based on Optimization I, which indicates that mouse flows are hardly estimated to be elephant flows even if their fingerprints are the same as elephant flows. Thus 16 bits are enough for an acceptable precision. For experiments on throughput, we ignore operations on the min-heap for the CM sketch, because we can only record flows whose estimated size is larger than a pre-defined threshold.

B. Metrics

Precision: Precision is defined as $\frac{\mathcal{C}}{k}$. Only $\mathcal{C}$ flows belong to the real top-k flows.

Average Relative Error (ARE): ARE is defined as $\frac{1}{|\Psi|}\sum_{f_i\in\Psi}\frac{|\hat{n}_i-n_i|}{n_i}$, where Ψ is estimated set of top-k flows, $\hat{n}_i$ is the estimated size of flow f_i, and n_i is the real size of flow f_i. ARE evaluates the error rate of the estimated flow size reported by the algorithm.

Average Absolute Error (AAE): AAE is defined as $\frac{1}{|\Psi|}\sum_{f_i\in\Psi}|\hat{n}_i-n_i|$, similarly to ARE.

Throughput: We perform insertions of all packets, record the total time used, and calculate the throughput. The throughput is defined as $\frac{N}{T}$, where N is the total number of packets, and T is the total measured time. We use Million of insertions per second (Mps) to measure the throughput.

C. Experiments on Precision

To achieve a head-to-head comparison, we use the same memory size for each algorithm, and use Hardware Parallel Version as our default version of HeavyKeeper. We perform the experiments for varying memory size and k on the campus and CAIDA datasets, and varying skewness on the synthetic datasets. For experiments of varying memory size, we set $k = 100$, and vary the memory from 10 to 50KB. For experiments of varying k, we set the memory size to 100KB, and vary k from 200 to 1000. For experiments of varying skewness, we set the memory size to 100KB, set $k = 1000$, and vary skewness from 0.6 to 3.0. The reason why we use such a small memory size is that when the memory is large, the gap of performance between ours and related works is very small. We argue that our algorithm significantly outperforms others when the memory size is tight.

[2] In network-wide measurement, sketches in different switches are often periodically sent to a collector for timely network traffic analysis. Each period is often small, for example, 10M packets.

[3] Assume there is a stream which has M distinct flows and let N be the total number of flows. Let f_i be the frequency of the i^{th} flow. The skewness γ of this stream refers to $f_i = \frac{N}{i^{\gamma}\delta(\gamma)}$, where $\delta(\gamma) = \sum_{j=1}^{M}\frac{1}{j^{\gamma}}$.

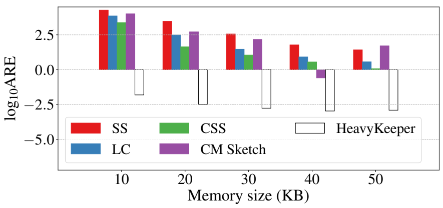


Fig. 10. ARE vs. memory size (CAIDA).

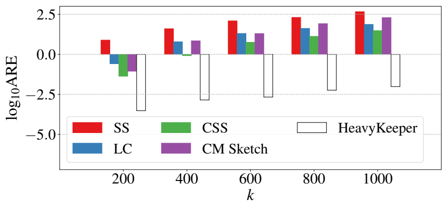


Fig. 11. ARE vs. k (Campus).

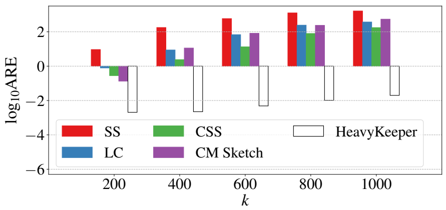


Fig. 12. ARE vs. k (CAIDA).

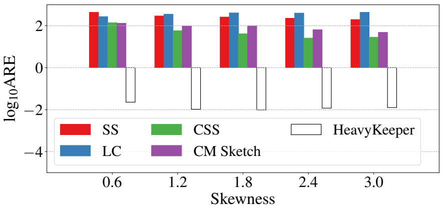


Fig. 13. ARE vs. skewness (Synthetic).

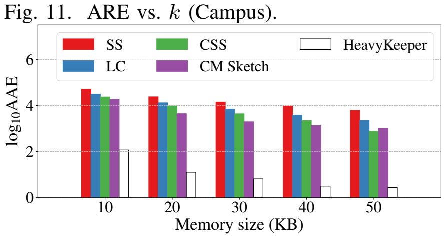


Fig. 14. AAE vs. memory size (Campus).

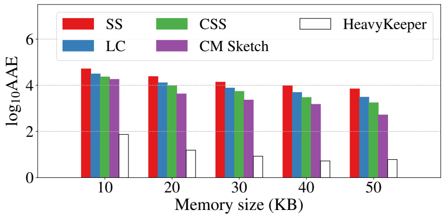


Fig. 15. AAE vs. memory size (CAIDA).

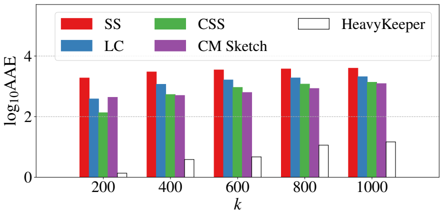


Fig. 16. AAE vs. k (Campus).

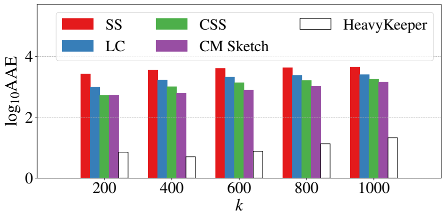


Fig. 17. AAE vs. k (CAIDA).

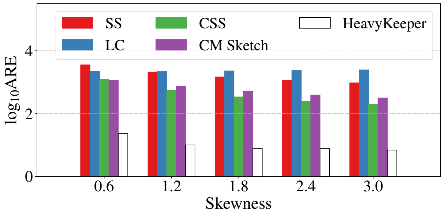


Fig. 18. AAE vs. skewness (Synthetic).

Precision vs. memory size: As shown in Figure 4, for the campus dataset, when memory size is 10KB, the precision of Space-Saving, Lossy counting, CSS, and CM sketch is respectively 10%, 11%, 19%, and 41%, while the one of HeavyKeeper is 82%. Furthermore, we find that the precision of HeavyKeeper reaches 100% for a memory size of 30KB, while the corresponding precision of Space-Saving, Lossy counting, CSS, and CM sketch is 27%, 39%, 49%, and 93%. This implies that HeavyKeeper has indeed much better precision than the other three algorithms. We find that Lossy counting is more accurate than Space-Saving. However, as will be mentioned later, Lossy counting is much slower than the other algorithms. For the CAIDA dataset (see Figure 5), we find that the precision of HeavyKeeper reaches 99.99% when memory size is larger than 20KB, while for Space-Saving, Lossy counting, CSS, and CM sketch, precision is respectively 18%, 33%, 34%, and 89% when memory size is 50KB.

Precision vs. k: As shown in Figure 6, for the campus dataset, as k becomes larger, the precision of HeavyKeeper stays high, while it degrades for other algorithms. For the campus dataset, as k becomes larger, the precision of HeavyKeeper is always higher than 95.9%, while that of Space-Saving, Lossy counting, CSS, and CM sketch reaches 32.7%, 44.1%, 50.1%, and 77.9% respectively when $k = 1000$. This happens for two main reasons: 1) larger k requires larger memory usage to store information about more flows; 2) as k increases, the difference of flow sizes among flows becomes smaller, so it is easy to mistake other flows for top-k flows. For the CAIDA dataset (Figure 7), we find that the precision of HeavyKeeper is always above 94%, while for Space-Saving, Lossy counting, CSS, and CM sketch, it is 26.6%, 37.1%, 44%, and 70% respectively when $k = 1000$.

Precision vs. skewness: As shown in Figure 8, the precision of HeavyKeeper reaches 99.99%. For all considered values of skewness, the precision of HeavyKeeper does not go below 94.9%, while the highest precision for Space-Saving, Lossy counting, CSS, and CM sketch is 46.8%, 41.3%, 74.5%, and 85.7%, respectively.

D. Experiments on AAE and ARE

In this section, we focus on the ARE and the AAE of the estimated frequency of reported top-k flows. We also conduct experiments with varying memory size, k, and skewness. The parameter settings are the same as in Section VI-C.

ARE vs. memory size: As shown in Figure 9, for the campus dataset, we find that the ARE of HeavyKeeper is smaller than 0.01 when memory size is larger than 20KB, while for Space-Saving, Lossy counting, CSS, and CM sketch, it is larger than 100. Furthermore, we find that the ARE of HeavyKeeper is between 100158 and 648291 times smaller than the one of Space-Saving, between 8450 and 78209 times smaller than the one of Lossy counting, between 945 and 49561 times smaller than the one of CSS, and between 279 and 226986 times smaller than the one of CM sketch. For the CAIDA dataset (see Figure 10), we find that the ARE of HeavyKeeper is between 21119 and 1190365 times smaller than the one of Space-Saving, between 2955 and 456275 times smaller than the one of Lossy counting, between 950 and 154047 times smaller than the one of CSS, and between 238 and 656145 times smaller than the one of CM sketch.

ARE vs. k: As shown in Figure 11, for the campus dataset, we find that the ARE of HeavyKeeper is between 25579 and 56791 times smaller than the one of Space-Saving, between 852 and 9312 times smaller than the one of Lossy counting, between 142 and 3132 times smaller than the one of CSS, and between 293 and 20370 times smaller than the of of CM sketch. For the CAIDA dataset (see Figure 12), we find that the ARE of HeavyKeeper is between 4506 and 121912 times smaller than the one of Space-Saving, between 383 and 23666 times smaller than the one of Lossy counting, between 137 and

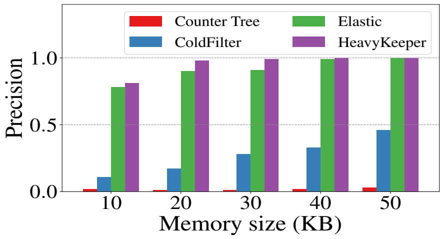


Fig. 19. Precision vs. memory size.

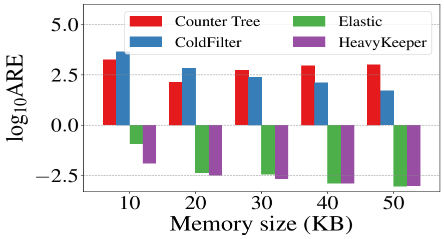


Fig. 20. Precision vs. ARE.

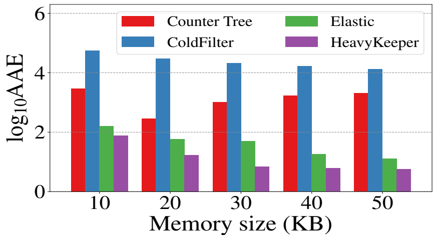


Fig. 21. AAE vs. AAE.

8816 times smaller than the one of CSS, and between 66 and 27290 times smaller than the one of CM sketch.

ARE vs. skewness: As shown in Figure 13, for all considered values of skewness, we find that the ARE of HeavyKeeper is between 15566 and 27829 times smaller than that of Space-Saving, between 11915 and 41575 times smaller than that of Lossy counting, between 2174 and 6099 times smaller than that of CSS, and between 3819 and 10080 times smaller than that of CM sketch.

AAE vs. memory size: As shown in Figure 14, for the campus dataset, we find that the AAE of HeavyKeeper is between 433 and 3013 times smaller than that of Space-Saving, between 267 and 1221 times smaller than that of Lossy counting, between 200 and 758 times smaller than that of CSS, and between 155 and 428 times smaller than that of CM sketch. When memory size is 50KB, the AAE of HeavyKeeper is only 2.73, confirming that the estimated flow sizes of almost all reported flows are accurate. For the CAIDA dataset (see Figure 15), we find that the AAE of HeavyKeeper is between 697 and 1810 times smaller than that of Space-Saving, between 421 and 928 times smaller than that Lossy counting, between 289 and 647 times smaller than the one of CSS, and between 86 and 284 times smaller than that of CM sketch.

AAE vs. k: As shown in Figure 16, for the campus dataset, we find that the AAE of HeavyKeeper is between 271 and 1382 times smaller than that of Space-Saving, between 142 and 346 times smaller than that of Lossy counting, between 93 and 196 times smaller than that of CSS, and between 74 and 318 times smaller than that of CM sketch. For CAIDA dataset (see Figure 17), we find that the AAE of HeavyKeeper is between 206 and 694 times smaller than that of Space-Saving, between 118 and 329 times smaller than that of Lossy counting, between 73 and 199 times smaller than that of CSS, and between 67 and 121 times smaller than that of CM sketch.

AAE vs. skewness: From Figure 18, we find that the AAE of HeavyKeeper is between 137 and 209 times smaller than that of Space-Saving, between 96 and 355 times smaller than that of Lossy counting, between 28 and 55 times smaller than that of CSS, and between 45 and 73 times smaller than that of CM sketch.

E. Compare with Recent Works

In this section, we compare our algorithm with recent works, including the Elastic sketch, Counter Tree and Cold Filter. Since HeavyGuardian is not applied in finding top-k elephant flows, we do not compare our algorithm with it. For the Elastic sketch and Cold Filter, the source codes are from their authors [36], [38]. We use Cold Filter with Space Saving to evaluate its performance, because the performance of Cold Filter with Space Saving is the best in that paper. For Counter Tree, we use the formulas derived from its author [39] to estimate frequencies of flows. We only report results for the campus dataset by varying the memory size. Here we set $k = 100$ and vary memory size from 10KB to 50KB.

Measuring precision: As shown in Figure VI-D, the precision of HeavyKeeper is much better than Counter Tree and Cold Filter. Next we explain the reason of the performance difference between our algorithm and others. For Counter Tree, it uses formulas to estimate frequencies of flows, which might cause a large error of accuracy. For Cold Filter, its key data structure is Space Saving, whose performance is worse than HeavyKeeper, and the filter takes up a certain amount of memory. For the Elastic sketch, it is a general data structures, while HeavyKeeper just focuses on finding top-k elephant flows. That is why HeavyKeeper is slightly better than the Elastic sketch.

Measuring ARE: As shown in Figure VI-D, the ARE of HeavyKeeper is the lowest compared with other recent works. Specifically, when the memory size is 10KB, the ARE of Counter Tree, ColdFilter and the Elastic sketch is $10^{3.2}$, $10^{3.6}$ and $10^{-0.9}$, respectively, while the one of HeavyKeeper is lower than $10^{-1.8}$. In a sense, HeavyKeeper could handle the situation in tight memory much better than other algorithms.

Measuring AAE: As shown in Figure VI-D, the AAE of algorithm algorithm is the lowest compared with other recent works. Specifically, when the memory size is 10KB, the AAE of Counter Tree, ColdFilter and the Elastic sketch is $10^{3.4}$, 10^{4} and $10^{2.1}$, respectively, while the one of HeavyKeeper is lower than $10^{1.9}$. As the memory size increases, the AAE of algorithm is always the lowest compared with other algorithms.

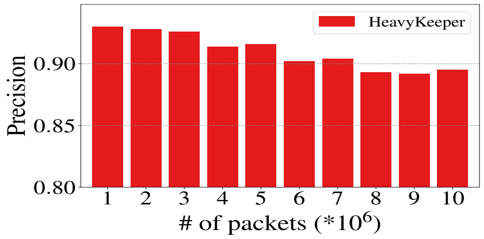


Fig. 22. Precision vs. # of packets.

F. Performance on Very Big Dataset

We also conduct experiments on very big dataset. We set $k = 1000$ and the memory size to 100KB. For every 10M packets, we report top-k elephant flows and evaluate the precision by comparing with real top-k elephant flows. As shown in Figure 22, as the total number of packets increases, the precision slightly reduces. However, we can obverse that the precision still reaches a high value when the total number of packets is 100M.

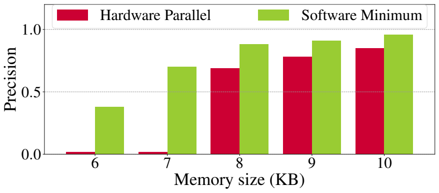


Fig. 23. Precision vs. memory size.

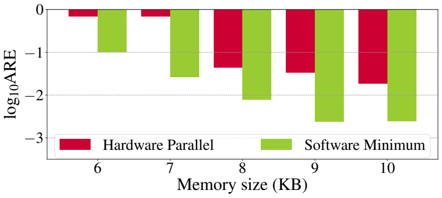


Fig. 24. ARE vs. memory size.

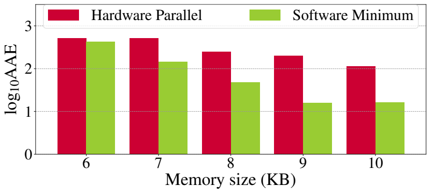


Fig. 25. AAE vs. memory size.

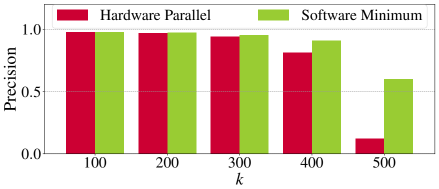


Fig. 26. Precision vs. k.

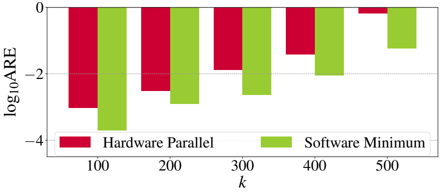


Fig. 27. ARE vs. k.

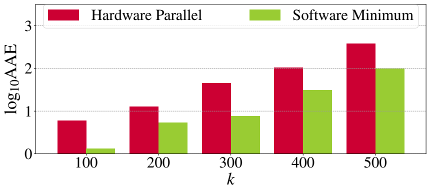


Fig. 28. AAE vs. k.

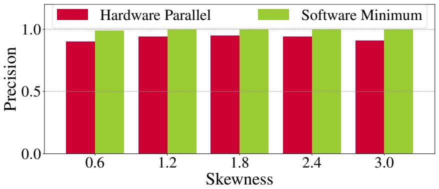


Fig. 29. Precision vs. skewness.

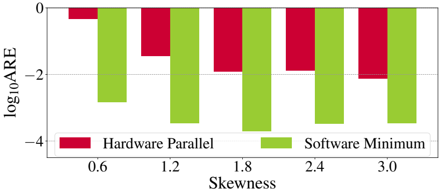


Fig. 30. ARE vs. skewness.

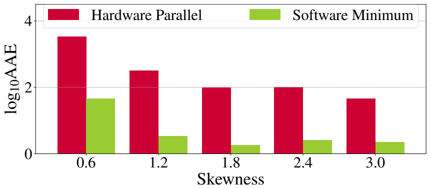


Fig. 31. AAE vs. skewness.

G. *Hardware Parallel Version vs. Software Minimum Version*

In this section, we compare Hardware Parallel Version with Software Minimum Version. We conduct experiments with varying memory size, k, and skewness. Due to the high accuracy of our algorithm, we set the smaller memory size to show the difference of performance between two versions clearly. Specifically, for experiments of varying memory size, we set $k = 100$, and vary the memory size from 6KB to 10KB; for experiments of varying k, we set the memory size to 30KB, and vary k from 100 to 500; for experiments of varying skewness, we set the memory size to 10KB and $k = 100$. Since the results are similar on CAIDA and campus datasets, we just show the performance of two versions on campus dataset.

Varying memory size: As shown in Figure 23, when memory size is 5KB or 6KB, the precision of Hardware Parallel Version is only 2%, and the reason behind is that there are only a few buckets, which cannot record all the largest k flows. On the other hand, the precision of Software Minimum Version achieves 38% and 70% when memory size is 5KB and 6KB, respectively, and the reason behind is that each flow has no duplicate when it is inserted into the hash table, and therefore the Software Minimum Version saves memory more efficiently. As shown in Figure 24, we find that the ARE of Software Minimum Version is between 5.8 and 14.6 times smaller than the one of Hardware Parallel Version. As shown in Figure 25, we find that the AAE of Software Minimum Version is between 1.1 and 8.2 times smaller than the one of Hardware Parallel Version.

Varying k: As shown in Figure 26, as k increases, the precision of Hardware Parallel Version decreases from 100% to 13%, while the Software Minimum Version still achieves 60% precision when $k = 1000$. As shown in Figure 27, we find that the ARE of Software Minimum Version is between 3.2 and 12.3 times smaller than the one of Hardware Parallel Version. As shown in Figure 28, we find that the AAE of Software Minimum Version is between 3.8 and 7.3 times smaller than the one of Hardware Parallel version.

Varying skewness: As shown in Figure 29, for all considered values of skewness, the precision of Software Minimum Version is always larger than the one of Hardware Parallel Version. As shown in Figure 30, we find that the ARE of Software Minimum Version is between 84 and 372 times smaller than the one of Hardware Parallel Version. As shown in Figure 31, we find that the AAE of Software Minimum Version is between 35 and 100 times smaller than the one of Hardware Parallel version.

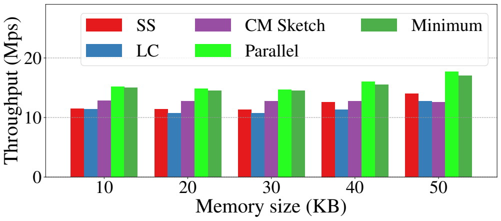


Fig. 32. Throughput vs. memory size.

H. *Experiments on Throughput*

We now turn to the throughput of the algorithms. We only report results for the campus dataset due to space limitations. We set $k = 100$, and vary memory size from 10KB to 50KB. Since our server of running experiments is much older than most of current ones, the throughput of experimental results might be slightly lower than the results in other papers.

Throughput vs. memory size: As shown in Figure 32, we find that the throughput of HeavyKeeper is always higher than other algorithms, and the throughput of HeavyKeeper of Hardware Parallel Version is slightly higher than the Software Minimum Version. Indeed, the average throughput of HeavyKeeper of Hardware Parallel Version and Software Minimum Version is 15.52Mps, 15.27Mps, respectively, while it is 12.15Mps, 11.34Mps, and 12.72Mps for Space-Saving, Lossy counting, and CM sketch. These results show that HeavyKeeper not only is more accurate than previous work, but also achieves higher throughput as well.

VII. Open vSwitch Deployment

In this section, we implement our HeavyKeeper algorithm on a software switch platform: Open vSwitch (OVS). We first present details of our implementation, and then present experimental results to show the performance of our algorithm running on Open vSwitch.

A. OVS Implementation

The OVS implementation of our HeavyKeeper algorithm consists of three components: 1) the modified OVS datapath, 2) the shared memory buffering flow IDs, and 3) the user-space program of HeavyKeeper processing flow IDs. For each incoming packet, it will be first inserted into the OVS datapath for forwarding. Besides, we modify the source codes of OVS datapath to parse the flow ID of the incoming packet, and then insert its flow ID into the shared memory (the shared memory is created initially). Finally, the user-space program will read the flow IDs from the shared memory, and process them as incoming packets.

In order to improve the performance of OVS, we integrate OVS with DPDK (Data Plane Development Kit). DPDK implements the datapath entirely in the user-space, and thus it eliminates the overhead of a context switch and memory copies between user-space and kernel-space.

B. OVS Evaluation

We use min-size packets to conduct experiments in OVS with 4 threads and 40G link to evaluate the throughput of HeavyKeeper and other algorithms. Besides, we also show the throughput of OVS without using any algorithm to show the impact of algorithms. We set the memory size to 50KB.

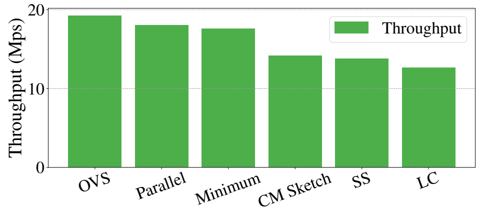


Fig. 33. Throughput on OVS platform.

As shown in Figure 33, the throughput of HeavyKeeper is near the original throughput of OVS. Specifically, the throughput of the original OVS is 19.22Mps, and that of HeavyKeeper of Hardware Parallel Version and Software Minimum Version is 18.03Mps, 17.62Mps, respectively. However, the throughput of CM sketch, Space-Saving, and Lossy Counting is 14.14Mps, 13.80Mps, and 12.64Mps, respectively. The results show that our HeavyKeeper algorithm has little impact to the performance of OVS, while other algorithms decrease the throughput significantly.

VIII. Conclusion

Finding the top-k elephant flows is a critical task for network traffic measurement. As the line rate increases, it is more and more challenging to design an accurate algorithm that achieves fast and constant speed. Existing algorithms for finding top-k flows cannot achieve high precision when traffic speed is high and memory usage is small. In this paper, we propose a novel data structure, called HeavyKeeper, which achieves a much higher precision on top-k queries and a much lower error rate on flow size estimation, compared to previous algorithms. The key idea of HeavyKeeper is that it intelligently omits mouse flows, and focuses on recording the information of elephant flows by using the exponential-weakening decay strategy. Our evaluation confirms that HeavyKeeper achieves 99.99% precision for finding the top-k elephant flows, while also achieving a reduction in the error rate of the estimated flow size by about 3 orders of magnitude compared to the state-of-the-art algorithms. We have released the source code of HeavyKeeper and all related algorithms at GitHub [42].

References

[1] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L. Uden, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," in *Proc. USENIX ATC*, 2018, pp. 909–921.

[2] A. Sivaraman, S. Subramanian, and et al., "Programmable packet scheduling at line rate," in *Proc. ACM SIGCOMM*, 2016.

[3] A. Feldmann, A. Greenberg, and et al., "Deriving traffic demands for operational ip networks: Methodology and experience," in *Proc. ACM SIGCOMM*, 2000.

[4] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *Proc. ACM IMC*, 2004.

[5] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 864–878, 2017.

[6] C. Hu, B. Liu, H. Zhao, and et al., "Disco: Memory efficient and accurate flow statistics for network measurement," in *IEEE ICDCS*, 2010, pp. 665–674.

[7] C. Hu, B. Liu, H. Zhao, K. Chen, Y. Chen, C. Wu, and Y. Cheng, "Disco: Memory efficient and accurate flow statistics for network measurement," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*. IEEE, 2010, pp. 665–674.

[8] H. C. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, and J. Xu, "A data streaming algorithm for estimating entropies of od flows," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 279–290.

[9] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, "Finding persistent items in data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 289–300, 2016.

[10] A. Kumar, J. Xu, and J. Wang, "Space-code bloom filter for efficient per-flow traffic measurement," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, 2006.

[11] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting bloom filter," *IEEE/ACM Transactions on Networking (TON)*, vol. 22, no. 4, pp. 1092–1105, 2014.

[12] S. Z. Kiss, E. Hosszu, J. Tapolcai, L. Rónyai, and O. Rottenstreich, "Bloom filter with a false positive free zone," *Proc. IEEE INFOCOM, Honolulu, HI, USA*, 2018.

[13] K. Mirylenka, G. Cormode, T. Palpanas, and D. Srivastava, "Conditional heavy hitters: detecting interesting correlations in data streams," *The VLDB Journal*, vol. 24, no. 3, pp. 395–414, 2015.

[14] J. H. Chang and W. S. Lee, "Finding recent frequent itemsets adaptively over online data streams," in *Proc. ACM SIGKDD*. ACM, 2003, pp. 487–492.

[15] Y.-L. Cheung and A. W.-C. Fu, "Mining frequent itemsets without support threshold: with and without item constraints," *IEEE TKDE*, vol. 16, no. 9, pp. 1052–1069, 2004.

[16] G. Salton and M. J. McGill, "Introduction to modern information retrieval," 1986.

[17] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang, "Top-k query processing in uncertain databases," in *Proc. IEEE ICDE*, 2007, pp. 896–905.

[18] Y. Zhang, B. Fang, and Y. Zhang, "Identifying heavy hitters in high-speed network monitoring," *Science China Information Sciences*, vol. 53, no. 3, pp. 659–676, 2010.

[19] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. SIGMOD 2016.*

[20] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, 2011.

[21] 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.

[22] C. Estan and G. Varghese, *New directions in traffic measurement and accounting*. ACM, 2002, vol. 32, no. 4.

[23] Y. Zhang, M. Roughan, W. Willinger, and L. Qiu, "Spatio-temporal compressive sensing and internet traffic matrices," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 267–278.

[24] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

[25] D. Maltz, "Unraveling the complexity of network management," 2009.

[26] Z. Li, F. Xiao, S. Wang, T. Pei, and J. Li, "Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 2, pp. 304–313, 2018.

[27] M. H. ur Rehman, C. S. Liew, A. Abbas, P. P. Jayaraman, T. Y. Wah, and S. U. Khan, "Big data reduction methods: a survey," *Data Science and Engineering*, vol. 1, no. 4, pp. 265–284, 2016.

[28] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *Proc. IEEE INFOCOM*, 2016.

[29] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. Springer ICDT 2005*.

[30] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Automata, languages and programming*, pp. 784–784, 2002.

[31] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. VLDB*, 2002, pp. 346–357.

[32] Z. Li, B. Chang, S. Wang, A. Liu, F. Zeng, and G. Luo, "Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things," *IEEE Transactions on Industrial Informatics*, 2018.

[33] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers." in *NSDI*, 2016, pp. 311–324.

[34] W. Feng and H. Mounir, "Matching the speed gap between sram and dram," in *Proc. IEEE HSPR*, 2008, pp. 104–109.

[35] E. Demaine, A. López-Ortiz, and J. Munro, "Frequency estimation of internet packet streams with limited space," *AlgorithmsESA 2002*, pp. 11–20, 2002.

[36] 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.

[37] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "Heavyguardian: Separate and guard hot items in data streams," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &amp; Data Mining*. ACM, 2018, pp. 2584–2593.

[38] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 741–756.

[39] C. Min and S. Chen, "Counter tree: A scalable counter architecture for per-flow traffic measurement," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–14, 2017.

[40] G. Einziger and R. Friedman, "Counting with tinytable: Every bit counts!" in *Proc. ICDCN 2016*.

[41] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proc. ACM IMC 2004*.

[42] "The source codes of heavykeeper and other related algorithms," https://github.com/papergitkeeper/heavy-keeper-project.

[43] "The caida anonymized internet traces 2016," http://www.caida.org/data/overview/.

[44] D. M. Powers, "Applications and explanations of zipf's law," in *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 1998, pp. 151–160.

[45] A. Rousskov and D. Wessels, "High-performance benchmarking with web polygraph," *Software: Practice and Experience*, 2004.

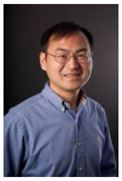
Tong Yang received his PHD degree in Computer Science from Tsinghua University in 2013. He visited Institute of Computing Technology, Chinese Academy of Sciences (CAS). Now he is a research assistant professor in Computer Science Department, Peking University. His research interests include network measurements, sketches, IP lookups, Bloom filters, sketches and KV stores. He published papers in SIGCOMM, SIGKDD, SIGMOD, SIGCOMM CCR, VLDB, ATC, ToN, ICDE, INFOCOM, *etc.*

Haowei Zhang is a sophomore student in Peking University, advised by Tong Yang. His research interests include network measurement and data stream processing systems. He has some publications in the area of networking and data stream processing.

Jinyang Li is a sophomore student in Peking University, advised by Tong Yang. Her research interests include network measurements, sketches, Bloom filters, data stream processing, and hash tables.

Junzhi Gong is a senior student in Peking University, advised by Tong Yang. His research interests include network measurement and data stream processing systems. He has some publications in the area of networking and data stream processing.

Steve Uhlig obtained a Ph.D. degree in Applied Sciences from the University of Louvain, Belgium, in 2004. From 2004 to 2006, he was a Postdoctoral Fellow of the Belgian National Fund for Scientific Research (F.N.R.S.). His thesis won the annual IBM Belgium/F.N.R.S. Computer Science Prize 2005. Between 2004 and 2006, he was a visiting scientist at Intel Research Cambridge, UK, and at the Applied Mathematics Department of University of Adelaide, Australia. Between 2006 and 2008, he was with Delft University of Technology, the Netherlands. Prior to joining Queen Mary, he was a Senior Research Scientist with Technische Universitt Berlin/Deutsche Telekom Laboratories, Berlin, Germany. Starting in January 2012, he is the Professor of Networks and Head of the Networks Research group at Queen Mary, University of London. Between 2012 and 2016, he was a guest professor at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.



Shigang Chen received the Ph.D. degrees in computer science from the University of Illinois at 1999. He is currently a Professor with the Department of Computer and Information Science and Engineering, University of Florida. He has authored over 160 peer-reviewed journal/conference papers. His research interests include computer networks, Internet security, wireless communications, and distributed computing. He served in various chair positions or as committee members for numerous conferences. He is an ACM Distinguished Member and a Distinguished Lecturer of the IEEE Communication Society. He was an Associate Editor for the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, and a number of other journals.

Xiaoming Li is a professor in computer science and technology and the director of Institute of Network Computing and Information Systems (NCIS) at Peking University, China. His current research interest is in search engine and web mining. He led the effort of developing a Chinese search engine (Tianwang) since 1999, and is the founder of the Chinese web archive (Web InfoMall).

Appendix

In this appendix, we first present the pseudo-code of HeavyKeeper in the Minimum Version, second prove that there is no over-estimation, and finally derive the formula of error bound of HeavyKeeper in the basic version.

Algorithm 2: Insertion process for finding top-k flows.

```
Input: A packet $\mathbb{P}_l$ belonging to flow $f_i$
$flag \leftarrow false$;
if $f_i \in min\_heap$ then
    $flag \leftarrow true$;
$min\_count \leftarrow A_1[h_1(f_i)].C$;
$min\_array \leftarrow 1$;
$first\_empty \leftarrow 0$;
$HeavyK\_V \leftarrow 0$;
$add\_flag \leftarrow false$;
for $j \leftarrow 1$ to $d$ do
    if $A_j[h_j(f_i)].FP = \mathbb{F}_i$ then
        if $flag = true$ or $A_j[h_j(f_i)].C < min\_heap.n_{min}$ then
            $A_j[h_j(f_i)].C++$;
            $add\_flag \leftarrow true$;
        $HeavyK\_V \leftarrow A_j[h_j(f_i)].C$;
        break;
    else
        if $A_j[h_j(f_i)].FP = null$ and $first\_empty = 0$ then
            $first\_empty \leftarrow j$;
        else
            if $A_j[h_j(f_i)].FP \neq null$ and
             $A_j[h_j(f_i)].C < min\_count$ then
                $min\_count \leftarrow A_j[h_j(f_i)].C$;
                $min\_array \leftarrow j$;
if $add\_flag = false$ then
    if $first\_empty > 0$ then
        $A_{first\_empty}[h_{first\_empty}(f_i)].FP \leftarrow \mathbb{F}_i$;
        $A_{first\_empty}[h_{first\_empty}(f_i)].C \leftarrow 1$;
        $HeavyK\_V \leftarrow 1$;
    else
        if $rand() < b^{-C}$ then
            $A_{min\_array}[h_{min\_array}(f_i)].C--$;
            if $A_{min\_array}[h_{min\_array}(f_i)].C = 0$ then
                $A_{min\_array}[h_{min\_array}(f_i)].FP \leftarrow \mathbb{F}_i$;
                $A_{min\_array}[h_{min\_array}(f_i)].C \leftarrow 1$;
                $HeavyK\_V \leftarrow 1$;
if $flag = true$ then
    $min\_heap[f_i] \leftarrow max(HeavyK\_V, min\_heap[f_i])$;
else
    if $min\_heap$ has empty buckets or
     $HeavyK\_V - n_{min} = 1$ then
        $min\_heap.insert(f_i)$;
```

A. Proof of No Over-estimation Error of HeavyKeeper

Theorem 4. *Let $n_i(t)$ be the real size of flow f_i at time t, and let $A_j[h_j(f_i)](t).C$ be the counter field of the mapped bucket of flow f_i in the j^{th} array at time t. If there is no fingerprint collision, then*

$$\forall j,t,\ \ A_j[h_j(f_i)](t).C \leqslant n_i(t) \tag{14}$$

Proof. When $t = 0$, no packet maps into this bucket, so $n_i(0) = 0$ and $A_j[h_j(f_i)](t).C = 0$. Therefore, the theorem holds at time 0. Let's now prove by induction that the theorem holds at any time.

When $t = 0$, the theorem holds.

If the theorem holds when $t = v$, let's prove that the theorem also holds when $t = v + 1$.

For HeavyKeeper in the basic version, there are three cases when $t = v + 1$:

Case 1: The new incoming packet is not mapped to bucket $A_j[h_j(f_i)]$. Then $n_i(v+1) = n_i(v)$ and $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Case 2: The new incoming packet belongs to flow f_i. Then $n_i(v+1) = n_i(v) + 1$ and $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C+1$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Case 3: The new incoming packet is mapped to bucket $A_j[h_j(f_i)]$ but does not belong to flow f_i. Then $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C$ or $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C - 1$, and $n_i(v+1) = n_i(v)$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Therefore, for HeavyKeeper in the basic version, for any time t,

$$A_j[h_j(f_i)](t).C \leqslant n_i(t)$$

For HeavyKeeper in the minimum version, Case 1 and Case 2 are the same. We just focus on Case 3.

Case 3 in the Minimum Version: The new incoming packet is mapped to bucket $A_j[h_j(f_i)]$ but does not belong to flow f_i. Suppose the new packet belongs to flow f_j ($f_i \neq f_j$). Among the other $d-1$ mapped buckets, if there is an empty bucket or there is a bucket holding the fingerprint $\mathbb{F}_j$, then HeavyKeeper does nothing to this bucket holding f_i. Thus, $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C$. If there is not, HeavyKeeper decays this bucket with a certain probability if and only if its counter is the first smallest among the counters of the d mapped buckets. Thus $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C$ or $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C - 1$.

Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$ holds in the minimum version. □

B. Error Bound of the Basic Version of HeavyKeeper

Definition A.1. *Given a small positive number ϵ, $Pr\{n_i - \hat{n}_i \geqslant \lceil \epsilon N \rceil\}$ ($n_i \geqslant \hat{n}_i$) represents the probability that the error of the estimated flow size $n_i - \hat{n}_i$ is larger than ϵN. If $Pr\{n_i - \hat{n}_i \geqslant \lceil \epsilon N \rceil\} \leqslant \delta$, the algorithm is said to achieve (ϵ,δ)-counting.*

(ϵ,δ)-counting is a metric to evaluate the error rate of the algorithm. Here HeavyKeeper is proved to achieve (ϵ,δ)-counting, showing that HeavyKeeper achieves a low error rate in estimating the sizes of top-k flows.

Theorem 5. *Let's assume that there is no fingerprint collision and once the fingerprint of an elephant flow is inserted into its mapped bucket, it is held there all the time. Let's focus on one single array of HeavyKeeper. Given a small positive number ϵ, and an elephant flow f_i whose size is n_i is held at that bucket,*

$$Pr\{n_i - \hat{n}_i \geqslant \lceil \epsilon N \rceil\} \leqslant \frac{1}{\epsilon w n_i (b-1)} \tag{15}$$

where w is the width of each array, N the total number of packets, and b the exponential base, M the total number of different flows.

Proof. Let's focus on the j^{th} array. Flow f_i is correctly reported, so at the end, the fingerprint of flow f_i is held in the $h_j(f_i)^{th}$ bucket of the j^{th} array. Let $I_{i,j,i'}$ be a binary random variable, defined as

$$I_{i,j,i'} = \begin{cases} 0 & (f_i = f_{i'}) \vee (h_j(f_i) \neq h_j(f_{i'})) \\ 1 & (f_i \neq f_{i'}) \wedge (h_j(f_i) = h_j(f_{i'})) \end{cases} \tag{16}$$

$I_{i,j,i'} = 1 \; iff$ different flows f_i and $f_{i'}$ are held at the same bucket in the j^{th} array. We define random variable $X_{i,j}$ as:

$$X_{i,j} = \sum_{i'=1}^{M} I_{i,j,i'} n_{i'} \tag{17}$$

$X_{i,j}$ represents the sum of the sizes of the flows held at the same bucket as flow f_i, except for the size of flow f_i itself. Assume that for each incoming packet, if it belongs to flow f_i, the counter field is incremented by 1; if not, the counter field is decayed with a certain probability. We have

$$n_i - X_{i,j} \leqslant A_j[h_j(f_i)].C \leqslant n_i \tag{18}$$

Specifically, if all packets that do not belong to flow f_i decay the counter field, then $A_j[h_j(f_i)].C = n_i - X_{i,j}$. If those packets do not decay the counter field, then $A_j[h_j(f_i)].C = n_i$. Let's define another random variable $P_{i,j,l}$. Among the $X_{i,j}$ packets defined above, $P_{i,j,l}$ is defined as the probability that the l^{th} packet decays the counter field. Therefore,

$$A_j[h_j(f_i)].C = n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l} \tag{19}$$

Given a small positive number ϵ, the following formula based on the Markov inequality holds

$$\begin{aligned} & Pr\{A_j[h_j(f_i)].C \leqslant n_i - \epsilon N\} \\ &= Pr\{n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l} \leqslant n_i - \epsilon N\} \\ &= Pr\{\sum_{l=1}^{X_{i,j}} P_{i,j,l} \geqslant \epsilon N\} \leqslant \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\epsilon N} \end{aligned} \tag{20}$$

Now let's focus on $E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$. Assume that all packets are uniformly distributed, for $\forall C$ satisfying $1 \leqslant C \leqslant n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$, we have the following formula:

$$Pr\{P_{i,j,l} = \frac{1}{b^{\mathrm{C}}}\} = \frac{1}{A_j[h_j(f_i)].C} = \frac{1}{n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})} \tag{21}$$

Let β be $n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$ for convenience. As a result,

$$\begin{aligned} E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) &= \sum_{l=1}^{E(X_{i,j})} E(P_{i,j,l}) \\ &= E(X_{i,j}) \sum_{\mathrm{C}=1}^{\beta} \frac{1}{b^{\mathrm{C}}} \frac{1}{\beta} = \frac{E(X_{i,j})}{\beta} \cdot \sum_{\mathrm{C}=1}^{\beta} \frac{1}{b^{\mathrm{C}}} \\ &= \frac{E(X_{i,j})}{\beta} \cdot \frac{\frac{1}{b}(1 - (\frac{1}{b})^{\beta})}{1 - \frac{1}{b}} \\ &\leqslant \frac{E(X_{i,j})}{n_i b} \cdot \frac{1 - (\frac{1}{b})^{n_i}}{1 - \frac{1}{b}} = \frac{E(X_{i,j})(1 - (\frac{1}{b})^{n_i})}{n_i(b-1)} \end{aligned} \tag{22}$$

Furthermore, for $E(X_{i,j})$, based on Equation 17,

$$E(X_{i,j}) = E(\sum_{i'=1}^{M} I_{i,j,i'} n_{i'}) \quad \leqslant \sum_{i'=1}^{M} n_{i'} E(I_{i,j,i'}) = \frac{N}{w} \tag{23}$$

Therefore, based on Equation 22,

$$E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) \leqslant \frac{N(1 - (\frac{1}{b})^{n_i})}{w n_i (b-1)} \leqslant \frac{N}{w n_i (b-1)} \tag{24}$$

then

$$\begin{aligned} Pr\{A_j[h_j(f_i)].C \leqslant n_i - \epsilon N\} &\leqslant \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\epsilon N} \\ &\leqslant \frac{N}{\epsilon N w n_i (b-1)} = \frac{1}{\epsilon w n_i (b-1)} \end{aligned}$$

Note that for an elephant flow f_i, n_i is very large, and $(\frac{1}{b})^{n_i} \approx 0$. The estimated size of f_i is the maximum value of $A_j[h_j(f_i)].C$, so we have

$$Pr\{n_i - \hat{n}_i \geqslant \lceil \epsilon N \rceil\} \leqslant Pr\{\hat{n}_i \leqslant n_i - \epsilon N\} \leqslant \frac{1}{\epsilon w n_i (b-1)}$$

□

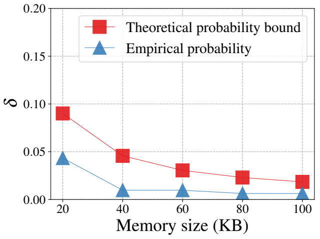


Fig. 34. Theoretical bound and empirical probability of HeavyKeeper ($\epsilon = 2^{-16}$).

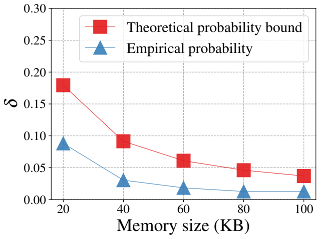


Fig. 35. Theoretical bound and empirical probability of HeavyKeeper ($\epsilon = 2^{-17}$).

To validate the correctness of this error bound, we conduct experiments on the dataset mentioned in Section VI-A. Here, we let $N = 10^7$, $\epsilon = 2^{-16}$ and 2^{-17}, and vary memory size from 20KB to 100KB. As shown in Figure 34 and Figure 35, the empirical probability of the basic version of HeavyKeeper is always lower than the theoretical probability bound, confirming the correctness of Theorem 5. Moreover, for the CSS algorithm, achieving such a (ϵ,δ)-counting requires at least $O(\epsilon^{-1})$ buckets (*i.e.*, $m = O(\epsilon^{-1})$), which requires a memory size much larger than 100KB. Therefore, HeavyKeeper is much more memory efficient than CSS.