Journal of Network and Computer Applications 224 (2024) 103853

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Research paper

ActiveGuardian: An accurate and efficient algorithm for identifying active elephant flows in network traffic

Bing Xiong [a], Yongqing Liu [a], Rui Liu [a], Jinyuan Zhao [b,*], Shiming He [a], Baokang Zhao [c], Kun Yang [d], Keqin Li [e]

[a] School of Computer and Communication Engineering, Changsha University of Science and Technology, Changsha 410114, PR China
[b] School of Information Science and Engineering, Changsha Normal University, Changsha 410199, PR China
[c] School of Computer, National University of Defense Technology, Changsha 410073, PR China
[d] School of Computer Science and Electronic Engineering, University of Essex, Wivenhoe Park, Colchester CO43SQ, UK
[e] Department of Computer Science, State University of New York at New Paltz, New York 12561, USA

ARTICLE INFO

MSC:
00-01
99-00

Keywords:
Network traffic measurement
Active elephant flows
Sketch
Adaptive Counter Update
Leapfrog Hashing

ABSTRACT

Active elephant flows, which indicate the real-time data transmission status, are of primary interest in network management and various applications. However, existing network measurement efforts mainly focus on finding elephant flows, and limited works on identifying active elephant flows suffer from low accuracy and heavy overheads. To address this issue, this paper proposes ActiveGuardian to identify active elephant flows with high accuracy, low memory, and high throughput. The key idea is to intelligently separate potential elephant flows from mice flows, and guard and report the information of active elephant flows in every time window. To obtain high accuracy, we devise a filtering module that adaptively filters unnecessary flows with low arrival rates, by applying an *Adaptive Counter Update* strategy. To achieve high memory utilization, we design a *Leapfrog Hashing* algorithm for the guarding module to effectively solve hash collisions. Lastly, we perform theoretical derivation on the false positive and the error bound of ActiveGuardian, and experimental evaluations on its performance with real network traffic traces. The experimental results show that ActiveGuardian achieves higher accuracy (99.65%) with identical memory sizes, and higher throughput (26.53Mps) than the state-of-the-art solutions.

1. Introduction

Network traffic measurement is indispensable to efficiently understand and manage network performance. Until now, extensive efforts have been carried out in this area to accurately identify elephant flows, where a flow's ID is usually defined as a combination of certain protocol header fields, and flow size is defined as the number of transmitted packets. Elephant flows contribute a large portion of network traffic, and have a great impact on network performance. Hence, it is significant to identify elephant flows, whose flow size exceeds a predefined threshold. Accurate identification of elephant flows plays a significant role in areas such as traffic engineering (Burnett et al., 2020), network security (Wang et al., 2011; Curtis et al., 2011), data mining (Mirylenka et al., 2015; Chang and Lee, 2003) and information retrieval (Davenport, 2012). For instance in network management, administrators can accurately estimate the demand for equipment and bandwidth based on the situation of elephant flows, allowing for rational capacity planning (Feldmann et al., 2001). In traffic engineering, traffic allocation and scheduling policies can be optimized by analyzing elephant flows to ensure priority and performance of critical applications or services and improve Quality of Service (QoS) (Sivaraman et al., 2016). In data mining, the analysis of elephant flows can identify the most influential features and attributes to optimize feature selection and dimensionality reduction, thereby improving the efficiency of model training and predictive accuracy (Cheung and Fu, 2004).

Elephant flow identification strives to collect historical statistical information of packet flows from the beginning of traffic measurement. Nevertheless, many applications primarily focus on the real-time state of packet flows, especially with a mass of packets arrived recently. For example in intrusion detection (Garcia-Teodoro et al., 2009), Intrusion Detection Systems (IDS) rely on real-time packet flow information to promptly spot suspicious activity and rapidly respond to threats before they can cause significant damage (Zhuang et al., 2021). In congestion control (Zhu et al., 2020; Liao et al., 2020), by identifying recently active elephant flows, network administrators can promptly detect congestion situations and quickly pinpoint the source of the problem

* Corresponding author.
E-mail address: zhaojy@csnu.edu.cn (J. Zhao).

https://doi.org/10.1016/j.jnca.2024.103853
Received 15 July 2023; Received in revised form 26 December 2023; Accepted 21 February 2024
Available online 24 February 2024

to implement congestion control measures. Additionally, bandwidth management and optimization of these active elephant flows can effectively predict and avoid potential congestion, enhancing network performance and stability. As for task offloading in network scenarios (Sarrar et al., 2012), it can alleviate the load on main processor to improve overall network performance, by offloading network tasks associated with recently active elephant flows to specialized hardware accelerators or dedicated network devices. Active elephant flows, which have a large number of packets frequently arrived within a certain time window, greatly influence the performance of network applications. Therefore, there is a pressing need to identify and report active elephant flows by time window.

Existing solutions for identifying active elephant flows, inspired by traditional elephant flow identification methods, usually employ sketch-based algorithms for efficient memory utilization. However, due to hash collisions in sketches and highly skewed flow size distribution, mice flows are probably mistreated as elephant flows, leading to low identification accuracy. To overcome this issue, several strategies have been proposed, including the majority vote algorithm (Boyer and Moore, 1991), exponential-weakening decay (Yang et al., 2018a, 2019), and pre-filtering stages (Garcia-Teodoro et al., 2009). While these solutions have made progress in separating elephant flows from mice flows, they still cannot support real-time queries and detect active elephant flows. More recent works, such as Clock-Sketch (Chen et al., 2021) and FastKeeper (Wang et al., 2021), have attempted to address these two challenges by employing sliding-window-based algorithms to accurately estimate flow rates of active elephant flows. However, it will incur significant memory overhead to record flow ID. In summary, existing solutions cannot simultaneously achieve high accuracy, low overhead, and high throughput in identifying active elephant flows.

In this paper, we propose a novel algorithm, named ActiveGuardian, to accurately identify active elephant flows. The key idea of ActiveGuardian is called *separate-mice-and-guard-active*. Specifically, ActiveGuardian consists of a filtering module intelligently separating suspected elephant flows from mice flows, and a guarding module guarding and reporting the information of active elephant flows in every time window. To ensure the persistent effectiveness of the filtering module, we propose an *Adaptive Counter Update* strategy that consistently filters unnecessary flows. To enhance the accuracy of the guarding module, we utilize a replacement policy evicting inactive flows that are wrongly treated as active elephant flows. To optimize the memory efficiency of the guarding module, we design a novel hashing algorithm called *Leapfrog Hashing* that provides storage locations for each incoming flow at all possible. Our main contributions can be summarized as follows:

- A novel algorithm called ActiveGuardian is designed for identifying active elephant flows, which constantly filters unnecessary flows and smartly guards elephant flows by time window, to simultaneously achieve high accuracy, low overhead, and high throughput.
- An *Adaptive Counter Update* strategy is proposed for the filtering module of ActiveGuardian, which persistently filters numerous mice flows and low-arrival-rate flows even under network traffic fluctuation, to improve the precision of subsequent elephant flow identification.
- A *Leapfrog Hashing* algorithm is devised for the guarding module of ActiveGuardian, which smartly find storage locations for all potential elephants via multiple mapping and kicking operations, to greatly reduce hash collision rates, enhancing accuracy and memory efficiency.
- We prove that ActiveGuardian can provide the precise estimation of active elephant flows, by theoretically deriving the false positive error rate of its filtering module and the error bound of its guarding module.
- We conduct extensive experiments with real network traffic traces, which indicates that ActiveGuardian achieves higher accuracy and throughput than the state-of-the-art.

The rest of the paper is organized as follows. Section 2 discusses related work regarding the identification of active elephant flows. In Section 3, we provide a detailed description of the ActiveGuardian algorithm. Section 4 presents the theoretical analysis of our algorithm's error bounds. In Section 5, we present our experimental setup and results, followed by a comparison with existing solutions. Finally, Section 6 concludes the paper.

2. Related work

Early works regarding elephant flow identification mainly focus on applying sketch-based algorithms to estimate the frequency of each flow in network traffic. Sketches, a kind of probabilistic data structure, have been widely employed for data stream summarization, achieving high memory efficiency at the expense of slightly less accuracy. A sketch is commonly manifested as a two-dimensional array, where each array contains multiple counters and an associated hash function that randomly maps incoming packets to the counters. The well-known CM Sketch (Cormode and Muthukrishnan, 2005) increases all mapped counters by 1 for an arrived packet, and returns the minimum value among these counters as the estimated size of its belonging flow. On this basis, CU Sketch (Estan and Varghese, 2003) only increases the smallest mapped counter by 1, which further improves the estimation accuracy of flow size. To achieve unbiased estimation, Count Sketch (Charikar et al., 2004) implements an additional hash function to determine whether the mapped counters are incremented or decremented, and returns their median as the query value. CMM Sketch (Deng and Rafiei, 2007) inserts packets to the counters in the same way as CM Sketch, but minuses noises during query operation. However, due to hash collisions, a mice flow is probably mistreated as an elephant flow when they share an identical counter. Moreover, it is challenging to allocate adequate memory for sketches ahead of time, due to unpredictable flow sizes in network traffic.

To adapt to highly skewed flow size distribution, there are several approaches to separate elephant flows from mice flows. CountMax (Yu et al., 2018) and MV-Sketch (Tang et al., 2019) apply the majority vote algorithm (MJRTY) (Boyer and Moore, 1991) to track the candidate elephant flow in each bucket. HeavyGuardian (Yang et al., 2018a) and HeavyKeeper (Yang et al., 2019) leverage a novel strategy, named exponential-weakening decay, to actively remove mice flows through decaying. Augmented Sketch (Roy et al., 2016) proposes a pre-filtering stage to identify the elephants and only permits the mice to sketch. Elastic Sketch (Yang et al., 2018b) monitors the elephants in the heavy part and evicts the mice to the light part by voting. Pyramid Sketch (Yang et al., 2017) develops a sketch framework to prevent counters from overflowing by automatically enlarging their sizes, which significantly improves the speed and accuracy of identifying elephant flows. Cold Filter (Zhou et al., 2018) designs a two-layer sketch, which captures mice flows in the first stage and elephant flows in the second stage, to accurately estimate both mice flows and elephant flows. Unfortunately, these solutions only support queries on the overall traffic from the beginning of the measurement, unable to provide real-time information of elephant flows by time window.

Closely related to our work, several notable studies have explored the identification of the most recent elephant flows (i.e., active elephant flows) by sliding window models. WCSS (Ben-Basat et al., 2016) extends Space Saving (Metwally et al., 2005) to keep track of the most recent elephant flows in a sliding window and supports constant time point queries. Sliding Sketch (Gou et al., 2020) presents a generic sketch framework and utilizes a scanning pointer to delete outdated flows. More recently, Clock-Sketch (Chen et al., 2021) adopts multiple counters to preserve information about all flows within a time window, and cleans information of inactive flows as quickly as possible. FastKeeper (Wang et al., 2021) applies a sliding-window-based algorithm to accurately estimate the flow rates of active elephants, and timely replace flows that have become small through bitmap-voting. However,

Table 1
Summary of works on active elephant flows identification.

Methods		Core idea	Shortcoming
Sketch-based algorithms	CU Sketch (Estan and Varghese, 2003)	Increasing the smallest mapped counter by 1	Potential mistreatment of mice flows as elephant flows due to hash collisions
	Count Sketch (Charikar et al., 2004)	Incrementing or decrementing the mapped counters by an additional hash function	
	CMM Sketch (Deng and Rafiei, 2007)	Subtracting noises during query operation	
Separating elephant flows from mice flows	CountMax (Yu et al., 2018) MV-Sketch (Tang et al., 2019)	Utilizing the majority vote algorithm for identifying elephant flows	Unable to provide real-time information of elephant flows by time window.
	HeavyGuardian (Yang et al., 2018a) HeavyKeeper (Yang et al., 2019)	Adopting the exponential-weakening decay strategy to actively remove mice flows	
	Augmented Sketch (Roy et al., 2016)	Designing pre-filtering stage to identify elephant flows	
	Elastic Sketch (Yang et al., 2018b)	Monitoring elephants in the heavy part and evicting mice to the light part by voting	
	Pyramid Sketch (Yang et al., 2017)	Automatically enlarging the size of counters	
	Cold Filter (Zhou et al., 2018)	Employing pre-filtering stage to filter mice flows	
Sliding window schemes	WCSS (Ben-Basat et al., 2016)	Extending Space Saving (Metwally et al., 2005) in a sliding window	Requiring to process a large number of mice flows, leading to low memory efficiency and poor estimation accuracy.
	Sliding Sketch (Gou et al., 2020)	Utilizing a scanning pointer to delete outdated flows.	
	Clock-Sketch (Chen et al., 2021)	Introducing CLOCK algorithm into item batch measurement	
	FastKeeper (Wang et al., 2021)	Applying a sliding-window-based algorithm to estimate flow rates of active elephants	
	BurstSketch (Zhong et al., 2021)	Employing the running track technique to efficiently select potential burst flows	

these sliding window schemes still need to process a large number of mice flows, resulting in low memory efficiency and poor accuracy of estimation. Further, BurstSketch (Zhong et al., 2021) first employs the running track technique to efficiently select potential burst flows, and then detects active elephant flows at the end of every time window by the snapshotting technique. Yet, this algorithm requires recording all flow IDs in the first stage with heavy memory overhead.

To provide a visualized overall comparison, Table 1 summarizes the aforementioned works on the identification of active elephant flows. As summarized from Table 1, Sketch-based algorithms suffer from potential mistreatment of mice flows as elephant flows. Existing solutions by separating elephant flows from mice flows are unable to effectively identify active elephant flows. Sliding window schemes still face the challenge of high memory overhead and low precision. In a nutshell, existing solutions cannot accurately identify active elephant flows while simultaneously achieving high throughput and memory utilization. To solve the above problem, we are motivated to propose a novel algorithm named ActiveGuardian, which achieves comprehensive performance promotion by intelligently separating mice flows and guarding active elephant flows.

3. The design of ActiveGuardian

In this section, we provide an identification scheme of active elephant flows, design the data structure and algorithm of ActiveGuardian, and present two optimization techniques to further improve the precision and memory efficiency of ActiveGuardian.

3.1. Scheme

To provide a comprehensive understanding on the operation of ActiveGuardian, we design an identification scheme of active elephant flows depicted in Fig. 1. The scheme consists of the following components: (a) Network devices such as routers and Software-Defined Networking (SDN) switches, typically performing packet forwarding in the network domain; (b) Data collector, responsible for capturing packet traffic from network devices and preprocessing data before being transmitted to subsequent processing and analysis components; (c) Flow identification component, namely ActiveGuardian, designed for identifying active elephant flows from the collected packet traffic; (d) Controller such as a central management host or SDN controller, which learns about active elephant flows, conducts further analysis and optimizes network management for better network application performance.

In Fig. 1, we deploy the data collector on SDN switches to capture incoming packet traffic. Subsequently, we run the ActiveGuardian to aggregate statistics on active elephant flows based on predefined parameters. The identification results are then sent to the SDN controller. The controller performs tasks such as traffic monitoring, performance management, and network optimization through data analysis. Additionally, ActiveGuardian can be deployed on the controller, where it identifies active elephant flows by receiving network traffic information from data collector. In practical deployments, it is crucial to appropriately configure devices, components, and algorithms, in accordance with specific network environment and requirements. By this way, the scheme is expected to process large-scale network traffic.

3.2. ActiveGuardian algorithm

Rationale: In this paper, we propose an algorithm called ActiveGuardian for identifying active elephant flows. ActiveGuardian is comprised of two modules. To filter unnecessary flows and select potential elephant flows efficiently, the first module adopts a compact two-counter design to estimate current flow size and record historical information of each incoming flow. In addition, we periodically reset all counters to prevent from filtering failures of the first module, accommodating the network traffic fluctuation. To identify active elephant flows accurately, the second module guards the potential elephant flows with a timely replacement policy, and reports active elephant flows at the end of every time window. Furthermore, we propose a novel hashing algorithm called *Leapfrog Hashing*, which spares storage locations for all incoming flows as possible, to optimize the space efficiency of the second module. The symbols frequently used in ActiveGuardian are summarized in Table 2.

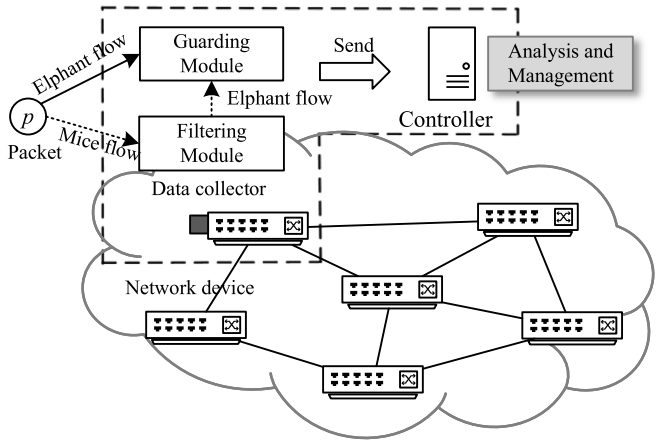


Fig. 1. Identification scheme of active elephant flows.

Table 2
Symbols in ActiveGuardian.

Symbol	Meaning
A_i	The ith bucket in the filtering module
B_i	The ith bucket in the guarding module
C_{cur}	The number of packets within a flow in current time window
C_{avg}	The weighted average flow size from the start of traffic measurement.
α	The filtering threshold
β	The guarding threshold
θ	The updating threshold of the filtering module
M	The number of buckets in the filtering module
N	The number of buckets in the guarding module
m	The number of cells in a bucket in the guarding module
γ	The updating threshold in adaptive counter update strategy

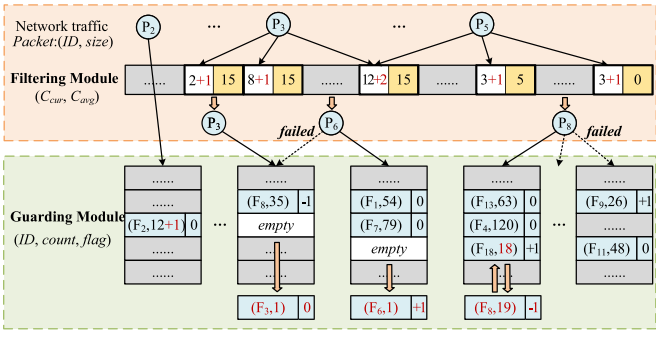


Fig. 2. The data structure of ActiveGuardian.

Data Structure: As shown in Fig. 2, ActiveGuardian consists of a filtering module evicting mice flows as well as low-arrival-rate flows and a guarding module identifying active elephant flows. The filtering module is a counter array with M buckets. There are k independent hash functions $h_1(.)$, $h_2(.)$,..., $h_k(.)$ mapping each flow to k buckets randomly. Each bucket has two counters: a current counter (C_{cur}) and a weighted average counter (C_{avg}). C_{cur} is used to estimate the number of packets within a flow in current time window, while C_{avg} is utilized to preserve the weighted average flow size from the start of traffic measurement. Note that we leverage a filtering threshold α for incoming flows to check whether they are potential elephant flows.

The guarding module is a bucket array $B[1]$, $B[2]$,..., $B[N]$ associated with one hash function $g(.)$. Each bucket consists of m cells. Each cell keeps the information of a flow: flow ID, count, and flag. The key is the flow ID, which consists of five-tuples: source IP address, destination IP address, source port, destination port, and protocol type. The count value is its estimated frequency in network traffic, i.e., the number of arrived packets within the flow. The flag indicates hashing offsets from the directly mapped bucket of the flow. Specifically, if the flag is equal to 0, the flow is preserved rightly in its hashed bucket. Otherwise, we can calculate its mapped position by adding its current position and hashing offset. Similarly, we leverage a guarding threshold β for all recorded flows to determine whether they are active elephant flows.

Algorithm 1: FilteringModuleInsert(Flow ID)

Input: The flow ID F_e of an incoming packet e; the filtering threshold α; the updating threshold θ; the packet counter C_{pkt}; the weight ω

```
1  if F_e is in B[g(F_e)] then
2      increment the frequency of F_e by 1;
3  else
4      for each i ∈ [1, k] do
5          A[h_i(F_e)].C_cur ← A[h_i(F_e)].C_cur + 1;
6          C_min^cur = min(C_min^cur, A[h_i(F_e)].C_cur);
7          C_min^avg = min(C_min^avg, A[h_i(F_e)].C_avg);
8      end
9      if C_min^cur ≥ α or C_min^avg ≥ α then
10         GuardingModuleInsert (F_e);
11     end
12     C_pkt ← C_pkt +1;
13     if C_pkt ≥ θ then
14         for each i ∈ [1, M] do
15             A[i].C_avg ← (A[i].C_avg + ωA[i].C_cur)/(1 + ω);
16             reset A[i].C_cur to 0;
17         end
18     end
19 end
```

Insertion: Initially, all fields in the two modules of ActiveGuardian are 0 or null. Given an incoming packet within the flow f, we first check whether f is already in the guarding module. If so, we increment the frequency of flow f by 1. Otherwise, we insert it into the filtering module. Below we show the details of our solution.

(1) Insertion of the filtering module: When inserting a packet within the flow f into the filtering module, ActiveGuardian first computes k hash functions to map f into k buckets $A[h_i(f)]$ ($1 \le i \le k$). Then $A[h_1(f)].C_{cur}$, $A[h_2(f)].C_{cur}$,..., $A[h_k(f)].C_{cur}$ are respectively incremented by 1, and applies different strategies for the following three cases:

Case 1: When $min\{A[h_i(f)].C_{cur}\} \ge \alpha$. In this case, the flow f is a potential elephant flow in current time window, and then we insert the packet into the guarding module, which corresponds to lines 9–10 in Algorithm 1.

Case 2: When $min\{A[h_i(f)].C_{avg}\} \ge \alpha$. In this case, the flow f is a potential elephant flow in the latest few time windows, and we also let the packet enter into the guarding module, manifested as lines 9–10 in Algorithm 1.

Case 3: When $min\{A[hi(f)]\}.C_{cur} < \alpha$ and $min\{A[hi\ (f)]\}.C_{avg} < \alpha$. In this case, the flow f is not a potential elephant flow by far, and then we discard the packet directly.

Note that we set a packet counter C_{pkt} tracking the number of arrived packets in current time window, along with an updating threshold θ determining whether the bucket will be updated. Specifically, after the insertion of an incoming packet, we increase the packet counter C_{pkt} by 1 (line 12 in Algorithm 1). When C_{pkt} reaches the updating threshold θ, we periodically accumulate each value in C_{cur} into C_{avg} by the weight ω, and subsequently resets the current counter C_{cur} to θ for each bucket of the filtering module (lines 13–15 in Algorithm 1).

(2) Insertion of the guarding module: When inserting a packet within the flow f into the guarding module, we first maps f to the bucket $B[g(f)]$ by the hash function $g(f)$ ($1 \le g(f) \le N$). For the mapped bucket, there are three cases as follows.

Algorithm 2: GuardingModuleInsert(Flow ID)

Input: The flow ID F_e of an incoming packet e; the guarding threshold β;

1 **if** F_e *is in* $B[g(F_e)]$ **then**
2 increment the frequency of e by 1;
3 **else if** $B[g(F_e)]$ *is empty* **then**
4 insert F_e into $B[g(F_e)]$ and set the frequency of F_e to 1;
5 **else**
6 look up two logical adjacent buckets of $B[g(F_e)]$;
7 **if** *an empty cell is found* **then**
8 insert F_e into the empty cell;
9 set $F_e.flag$ as the hashing offset of F_e;
10 **else**
11 find out the minimum flow F_{min} in $B[g(F_i)]$ and its two adjacent buckets;
12 **if** $F_{min}.Count < \beta$ **then**
13 replace F_{min} with F_e and set the frequency of F_e to $F_{min}.Count + 1$;
14 set $F_e.flag$ as the hashing offset of F_e;
15 **end**
16 **end**
17 **end**

Case 1: f is not in $B[g(f)]$, and the bucket still has empty cells. We insert f into an empty cell with the frequency of 1 according to lines 3–4 in Algorithm 2.

Case 2: f is in one cell in $B[g(f)]$. We just increase the corresponding frequency in the cell by 1 according to lines 1–2 in Algorithm 2.

Case 3: f is not in $B[g(f)]$, and the bucket has no empty cells. We initially look up the two logical adjacent $B[(g(f) + 1)\%N]$ and $B[(g(f) - 1)\%N]$. If there exists an empty cell, then we insert f into the cell and update the flag according to the hashing offset (lines 7–8 in Algorithm 2). Otherwise, we evict the flow with the minimum frequency in the mapped bucket and two logical adjacent buckets, and then insert f into the corresponding cell. Particularly, we just increase the count by 1, and set the flag of f with its hashing offset (lines 11–14 in Algorithm 2).

Algorithm 3: IdentifyAndUpdate()

Input: the last report time T_{last}; the current time T_{cur}; the guarding threshold β; the time window size W;

1 **if** $T_{cur} - T_{last} \geq W$ **then**
2 **for** *each* $i \in [1, N]$ **do**
3 **if** $B[i].Count \geq \beta$ **then**
4 **return** $B[i].ID$ and $B[i].Count + \alpha$;
5 **end**
6 clear $B[i]$ to empty;
7 **end**
8 **end**
9 $T_{last} \leftarrow T_{cur}$;

The identification of active elephant flows: ActiveGuardian constantly monitors potential elephant flows, and reports active elephant flows illustrated in Algorithm 3. We define time window as a fixed quantity of measured data packets, and denote the size of time window as W. At the end of every time window, we first examine the frequencies of guarded elephant flows in each bucket of the guardian module. If their frequencies are higher than the guarding threshold β, ActiveGuardian identifies them as active elephant flows, and reports their flow sizes. Finally, in preparation for next time window, we empty all buckets of the guarding module and set current time to the last report time.

A running example: Fig. 2 shows a running example of flow processing by ActiveGuardian. In this example, in the filtering module, given a bucket with (2, 15), where 2 represents the current counter (C_{cur}), and 15 is the weighted average counter (C_{avg}). In the guarding module, given a cell in a bucket with (F_2, 12), where F_2 is the flow ID, and 12 represents F_2's frequency. Suppose the filtering threshold $\alpha = 10$, and the guarding threshold $\beta = 50$. (1) To insert P_2 within the flow F_2, we find it in the guarding module, so we just increment $F_2.Count$ by 1. (2) To insert P_5 within the flow F_5, we do not find it in the guarding module, so we insert it into the filtering module and increment $A[h_j(F_5)].C_{cur}$ $(1 \leq j \leq k)$ by 1. (3) To insert P_3 within the flow F_3, we do not find it in the guarding module, so we insert it into the filtering module similarly to P_5. After the increment, we find that the $min(A[h_j(F_3)].C_{avg})$ $(1 \leq j \leq k)$ reaches the filtering threshold α and there is an empty cell in the guarding module. Then we insert F_3 with the frequency of 1. (4) To insert P_6 within the flow F_6, which passes through the filtering module, we observe that the mapped bucket of the guarding module is full, so we try to find an empty cell in its two logical adjacent buckets. Then we insert it into the guarding module with the frequency of 1 and set its flag to +1. (5) To insert P_8 within the flow F_8 admitted by the filtering module, we do not find any empty cell in the mapped bucket as well as its two adjacent buckets in the guarding module, so we replace the minimum flow F_{18} with F_8. Meanwhile, we set the frequency to 19 and the flag to −1; Therefore, we report F_1, F_7, F_{13}, and F_4 as active elephant flows, since their frequencies reach the guarding threshold β at the end of time window. Eventually, we clean all the buckets in the guarding module.

3.3. Optimization 1: Adaptive Counter Update strategy

In filtering module, we update all counters after inserting a fixed number of packets, irrespective of packet arrival rates. This leads to the steady operation of the filtering module even under network traffic fluctuations. However, it is challenging to determine an appropriate filtering threshold for a given number of packets in each updating period, due to variable flow size distributions even in a specific network scenario. In particular, a low filtering threshold may result in misidentifying potential elephant flows, while a high one probably leads to the omission of real elephant flows. The filtering threshold is typically configured based on average flow size varying with network traffic fluctuations. It is worth noting that the filtering threshold determines the size of each counter in the filtering module. Precisely, the counter size generally rounds up to the logarithm base two of the filtering threshold (Cormode and Muthukrishnan, 2005; Estan and Varghese, 2003; Charikar et al., 2004). Consequently, the counter size has to be set by the maximum of possible filtering thresholds under different states of network traffic, which incurs unnecessary memory overhead.

To address the above issue, we propose a novel strategy named *Adaptive Counter Update*, which supports arbitrary configurations of the counter size, and adapts to network traffic fluctuations. In this strategy, we set the filtering threshold as the maximum of each counter with an arbitrary size, to ensure optimal memory utilization. Then we update all counters when the proportion of overflowed counters reaches a predetermined threshold, to steadily filter mice flows with lower-ranking number of packets in flow size distribution. By this way, the counter size barely influences the accuracy of filtering mice flows, and can be flexibly configured in accordance with the constraints of network applications on the memory space of traffic measurement. In addition, the threshold limits the number of packets entering the filtering module, resulting in that the filtering accuracy is independent of packet arrival rates.

Fig. 3 presents a typical instance of the proposed *Adaptive Counter Update* strategy. Suppose the counter size is 4 bits with the filtering threshold α is 15, the weight ω is 1, and the updating threshold γ is 60%. When inserting an incoming packet, we increment each current counter C_{cur} in its mapped buckets by 1, and monitor the proportion of

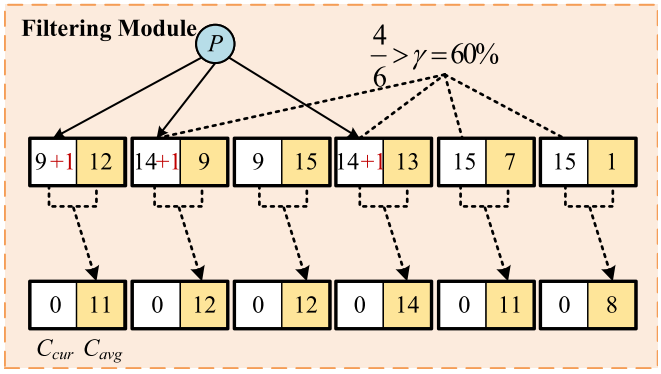


Fig. 3. The Adaptive Counter Update strategy.

overflowed counters to decide if an update is necessary. In this instance, 4 out of 6 current counters have overflowed, already surpassing the updating threshold γ. Therefore, we update each average weighted counter C_{avg} by average weighted accumulation, and reset all current counters to 0. Taking the first bucket for example, the C_{cur} is 10, and the C_{avg} is 12. We update the C_{avg} through computing the weighted average of 10 and 12 with the weight ω. Thus, we set C_{avg} to 11 and subsequently empty C_{cur}.

3.4. Optimization 2: Leapfrog Hashing algorithm

In guarding module, we guard and track potential elephant flows in each bucket, so as to accurately identify and report active elephant flows at the end of each time window. However, a large number of flows passing through the filtering module will result in high hash collision rates when network traffic surges. This is likely to further cause the omission of real active elephant flows, thereby degrading identification accuracy. To alleviate this problem, we propose a novel hashing algorithm called *Leapfrog* to promote memory utilization. The algorithm makes room for each incoming flow via iterative kicking operations, striving to evict an inactive flow while retaining active elephant flows. Specifically, if all candidate positions for an incoming flow are occupied, we select a flow with minimum packets from the two logical adjacent buckets of its mapped position, and kick it into its adjacent bucket away from the mapped position. Subsequently, we repeatedly kick the minimum flow in current bucket into its adjacent bucket in same direction, until we find an empty location for it or an inactive flow for replacement.

Query: For an incoming packet within the flow f, we query the guarding module to check whether f has already been recorded. In particular, we initially map f into a bucket in the guardian module by hashing, and look up the mapped bucket along with its two logical adjacent buckets. If the lookup succeeds to match the flow f, we increment its frequency by 1.

Insertion: Given a new flow f passing through the filtering module, we insert f into the guarding module. Specifically, we first locate its mapped bucket and two logical adjacent buckets. Then we insert the flow f into these buckets with three cases:

Case 1: There exists an empty cell in the mapped bucket. We insert f into the empty cell, and set its frequency to 1.

Case 2: There exists an empty cell in the two logical adjacent buckets. We insert f into the empty cell with the frequency of 1, and set its flag with its hashing offset.

Case 3: There exists no empty cell in the three buckets. We choose the minimum flow with the frequency below the guarding threshold β in the three buckets for replacement. If no flow can be replaced, we select a flow from a logical adjacent bucket to be kicked out, and insert f. For the kicked flow, we continue to look for an empty bucket

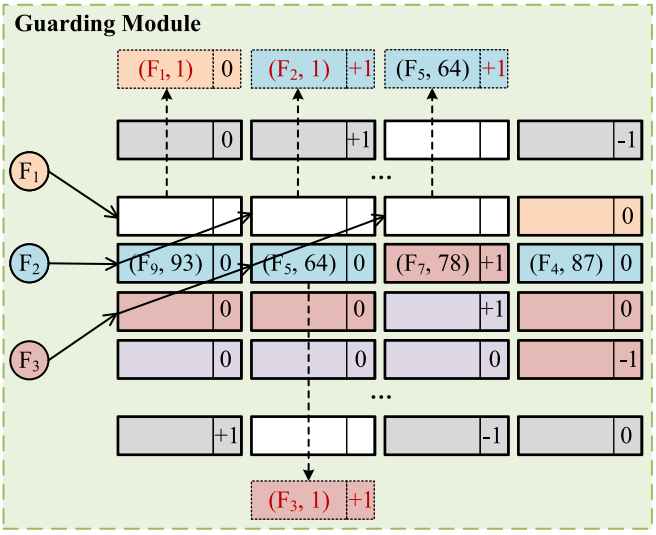


Fig. 4. A typical example of the Leapfrog hashing algorithm.

Table 3
Symbols in mathematical analysis.

Symbol	Meaning
L	The ith bucket of the filtering module
M	The ith bucket of the guarding module
$\|\|\bar{f}\|\|$	The average number of packets in mice flows
G	The number of flows inserted into the guarding module
σ	The average number of packets per flow of G
g	The number of flows carrying more than β packets
$\bar{f}$	The number of packets in the flow f
$\hat{f}$	The estimated number of packets in the flow f

to insert, or select the minimum flow with the frequency below the guarding threshold β to replace, in the adjacent bucket in the kicking direction. After the insertion or replacement, we update its flag with its new hashing offset. If there are no empty cell and no flow can be replaced, we iteratively perform the above operations until the number of kicking operations reaches a preset threshold.

Fig. 4 displays a typical example of the proposed *Leapfrog Hashing* algorithm. Suppose the guarding threshold β is 50. (1) To insert F_1, we find an empty cell in its mapped bucket, and insert it with the frequency of 1. (2) To insert F_2, we do not find an empty cell in its mapped bucket, but find one in an adjacent bucket. Thus, we insert F_2 with the frequency of 1 and set its flag to +1. (3) To insert F_3, we do not find any empty cell for insertion or flow for replacement, so we kick out F_5 with the minimum frequency 64 to its adjacent bucket and increment its flag by 1. Subsequently, we insert F_3 to the cell that has just been emptied, and set its flag to +1.

4. Mathematical analysis

In this section, we present the theoretical analysis of ActiveGuardian, including false positive error rate of the filtering module and error bound of the guarding module. The symbols frequently used in mathematical analysis are summarized in Table 3.

4.1. False positive error rate of filtering module

Theorem 1. *Suppose there are L elephant flows and S mice flows in network traffic. The filtering module consists of M buckets, with k independent hash functions. Given a flow entering the filtering module, the false positive error rate P_{FPR} can be estimated as follows:*

$$P_{FPR} = \left[1 - \left(1 - \frac{1}{M}\right)^{L} + 1 - \sum_{i=0}^{\lambda-1} \left(\frac{1}{M}\right)^{i} \left(1 - \frac{1}{M}\right)^{S-i}\right]^{k} \quad (1)$$

where $\lambda = \left\lceil \frac{\alpha}{\|\bar{f}\|} \right\rceil$, *$\alpha$ is the filtering threshold and* $\|\bar{f}\|$ *represents the average number of packets in mice flows.*

Proof. For an incoming mice flow f, it is first mapped to k buckets in the filtering module through k hash functions. If all k buckets reach the filtering threshold α, this mice flow will be erroneously classified as an elephant flow, which constitutes a false positive error. There are two cases for each bucket to reach the filtering threshold: (1) with at least one mapped elephant flow, or (2) with at least λ mapped mice flows. We will proceed to respectively determine the probabilities of these two cases.

Suppose that all flows randomly map into M buckets, the probability of any bucket not mapped by an elephant flow is 1-1/M. Since there are L elephant flows in total, the probability of any bucket mapped by at least one elephant flow can be derived as:

$$P_L = 1 - \left(1 - \frac{1}{M}\right)^L \tag{2}$$

Similarly, the probability of any bucket not mapped by a mice flow is 1-1/M. Since there are a total of S mice flows, the probability of any bucket mapped by i ($0 \le i \le S$) mice flows can be deduced as:

$$P_i = \left(\frac{1}{M}\right)^i \left(1 - \frac{1}{M}\right)^{S-i} \tag{3}$$

Therefore, we can compute the probability that any bucket is mapped by at least λ mice flows as:

$$P_S = 1 - \sum_{i=0}^{\lambda-1} P_i = 1 - \sum_{i=0}^{\lambda-1} \left(\frac{1}{M}\right)^i \left(1 - \frac{1}{M}\right)^{S-i} \tag{4}$$

In summary, the probability that any bucket reaches the filtering threshold can be expressed as $P_L + P_S$. Consequently, we can obtain the probability that all k buckets mapped by a mice flow reach the threshold, that is, the false positive error rate as:

$$\begin{aligned} P_{FPR} &= \left(P_L + P_S\right)^k \\ &= \left[1 - \left(1 - \frac{1}{M}\right)^L + 1 - \sum_{i=0}^{\lambda-1} \left(\frac{1}{M}\right)^i \left(1 - \frac{1}{M}\right)^{S-i}\right]^k \end{aligned} \tag{5}$$

Theorem holds. □

4.2. Error bound of the guarding module

Theorem 2. *Assume that there are G flows passing through the filtering module and inserted into the guarding module, and the average number of packets per flow is σ. Let N be the total number of buckets in the guarding module, m be the number of cells in each bucket, β be the packet number threshold of the guarding module. Suppose there are g flows each of which carries more than β packets. For a flow f with the number of packets $\tilde{f}$, let $\hat{f}$ be its estimated frequency, we can bound the error expectation of its frequency estimation in the guarding module as:*

$$E(|\tilde{f} - \hat{f}|) \le \frac{\beta}{\sigma - \alpha} \left(\frac{e^{\lambda} \lambda^m}{m!}\right)^3 \frac{G - g}{GNm} \tag{6}$$

where $\lambda = G/N$.

Proof. To determine the error in estimating the frequency of any flow f, we first analyze all cases that have an impact on the flow for an arrived packet entering the guarding module as follows.

Case 1: The packet resides in the flow f. In this case, we locate the flow in the guarding module, and increment its count value by 1. Thus, there is no error in estimating the frequency of f, i.e., $|\tilde{f} - \hat{f}| = 0$.

Case 2: The packet belongs to a new flow f', and there is no empty cell in its mapped bucket and adjacent buckets, where f is the minimum flow with a frequency lower than the guarding threshold β. In this case, the flow f will be replaced by f' and its estimated frequency will become zero. Hence, the estimation error of its frequency equals to its count value, i.e., $|\tilde{f} - \hat{f}| = f.Count$.

Case 3: The packet is affiliated to a new flow f', and there is no vacancy in its mapped bucket and adjacent buckets, where all flows have higher frequencies than the guarding threshold β and f is the minimum flow among them. In this case, the flow f will be kicked out to a new cell in one of its adjacent buckets for preserving f'. Therefore, there is also no error in estimating the frequency of f, i.e., $|\tilde{f} - \hat{f}| = 0$.

According to the above analysis, we can conclude that the estimation error of flow frequency primarily stems from flow replacements in case 2. Then, we analyze the probability of case 2 with the following conditions: (1) A new flow f' is arrived; (2) There is no empty cell in its mapped and adjacent buckets; (3) The flow f is preserved in these buckets $B[g(f')]$, $B[(g(f') + 1)\%N]$, and $B[(g(f') - 1)\%N]$; (4) The flow f is the minimum one in the buckets. (5) The count value of the flow f is lower than the guarding threshold β.

For a flow f', it is classified as a new flow only when its first packet comes into the guarding module. Since the average number of packets per flow is σ, and a flow must first reach the filtering threshold α before getting into the guarding module. Therefore, we can derive the probability of the condition 1 as the ratio of the number of flows to the total number of packets entering the guarding module in (7).

$$P_1 = \frac{G}{G(\sigma - \alpha)} = \frac{1}{\sigma - \alpha} \tag{7}$$

Given that each bucket operates independently, the probability of all three buckets having no empty cells can be considered as a classic bin packing problem. Then, we model this problem with the Poisson stream, where the parameter $\lambda = G/N$ represents the average number of flows in each bucket. Accordingly, the probability that a bucket is full can be expressed as:

$$P_{full} = \frac{e^{-\lambda} \lambda^m}{m!} \tag{8}$$

Consequently, we can calculate the probability of the condition 2 that all three buckets are full in (9).

$$P_2 = P_{full}^3 = \left(\frac{e^{-\lambda} \lambda^m}{m!}\right)^3 \tag{9}$$

Owing to that each flow is randomly mapped across N buckets by hashing, all buckets have the equal probability $1/N$ to accommodate the flow f. Therefore, we can get the probability of the condition 3 that the flow f is preserved in the three buckets $B[g(f')]$, $B[(g(f') + 1)\%N]$, and $B[(g(f') - 1)\%N]$ in (10).

$$P_3 = \frac{3}{N} \tag{10}$$

Since each bucket contains m cells, we can obtain the probability of the condition 4 that the flow f is the minimum one in the three buckets in (11).

$$P_4 = \frac{1}{3m} \tag{11}$$

Due to that there are G flows entering into the guarding module and g flows each of which carries more than β packets, we can compute the number of flows with a frequency lower than β as G-g, and the probability of the condition 5 that the count value of flow f is lower than the guarding threshold β in (12).

$$P_5 = \frac{G - g}{G} \tag{12}$$

By integrating the above probabilities of 5 conditions in (7), (9), (10), (11) and (12), we can infer the probability of flow replacements in (13).

$$P_{replace} = P_1 P_2 P_3 P_4 P_5 = \frac{1}{\sigma - \alpha} \left(\frac{e^{\lambda} \lambda^m}{m!}\right)^3 \frac{G - g}{GNm} \tag{13}$$

Since the frequency of a replaced flow is lower than the guarding threshold β, we can conclude that the error in estimating the frequency

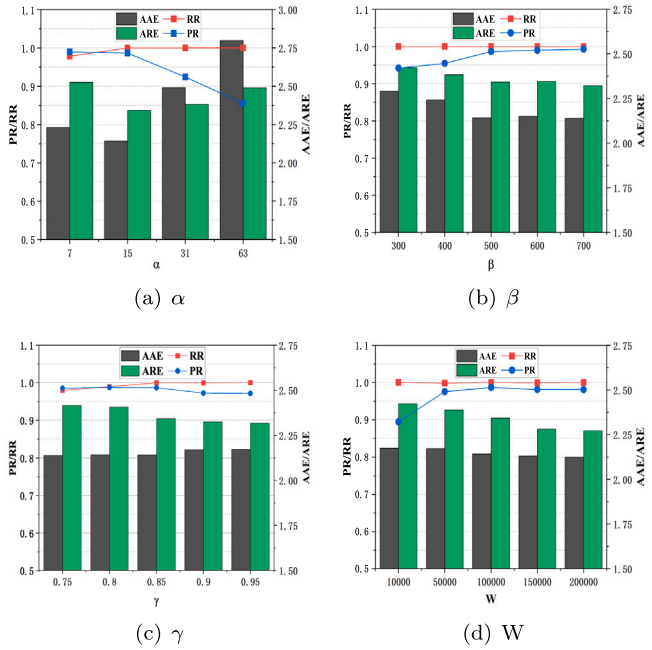


Fig. 5. Parameter settings.

of f satisfies $|\tilde{f} - \hat{f}| < \beta$. By combining with (13), we can eventually bound the error expectation of flow frequency estimation in the guarding module as:

$$E(|\tilde{f} - \hat{f}|) \leq \beta P_{replace} = \frac{\beta}{\sigma - \alpha} \left(\frac{e^{\lambda} \lambda^{m}}{m!} \right)^{3} \frac{G - g}{GNm} \tag{14}$$

Theorem holds. □

5. Experiments

In this section, we introduce our experimental setup, metrics, and parameter settings, and validate the superiority of ActiveGuardian by comparative experiments with existing solutions.

5.1. Experimental setup

Platform: Our testbed is comprised of 2 servers connected directly to each other. Each server is an HP ProLiant DL380 Gen10, equipped with two 10-core Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz and 128 GB RAM. One server assumes the role of the sender, while the other functions as the receiver. ActiveGuardian is implemented on the receiver side. Each server is outfitted with one Broadcom NetXtreme E-Series dual-port 100GbE NIC.

Dataset:

MAWI Dataset: The MAWI dataset was collected from daily traces at the transit link of WIDE to the upstream ISP (MAWILab, 2019). As for each packet in the dataset, we extract its packet size, timestamp, and a standard 5-tuple (IP address and port number for source and destination, and protocol type) utilized to construct flow ID. We select 20M packets in total, belonging to 4.6M flows with 6579 active elephants by setting the window size as 100K items.

WebDocs Dataset: The WebDocs dataset is downloaded from the website (dataset, 2005). This dataset was created with a collection of crawled web pages. Each item in the dataset is 4-byte long, representing the number of distinct items in a web page. We select 20M items with 8761 active elephants by setting the window size as 100K items.

Implementation: We have implemented BurstSketch (Zhong et al., 2021), FastKeeper (Wang et al., 2021), HeavyKeeper (Yang et al., 2019), ASketch (Roy et al., 2016), WCSS (Ben-Basat et al., 2016) and our ActiveGuardian in C++. For BurstSketch, the size of Stage 1 and Stage 2 are determined by the memory size. For HeavrKeeper, the number of arrays is 3, and the width of each array is determined by the memory size. For ASketch, the size of the filter is set from 0.1 kB to 12 kB, the sketch occupies the rest memory size. For WCSS, the size is also determined by the memory size. In our algorithm, we set the number of hash functions k to 3, the updating threshold γ to 0.85, the filtering threshold α to 15, the guarding threshold β to 500, and the size of ActiveGuardian depends on the memory size. The counter field in the filtering module and the guarding module are 4 bits and 16 bits respectively. In particular, some of the parameters mentioned above are based on experimental evaluations to obtain the best performance.

For each dataset, we read 20M packets. We set the length of the window W = 100K, and vary the memory size between 10 kB and 50 kB. When the frequency of a flow in the present time window is more than 500, we consider it as an active elephant flow. We compare the precision rate, recall rate, ARE, AAE and throughput of the above solutions under the same memory size.

5.2. Metrics

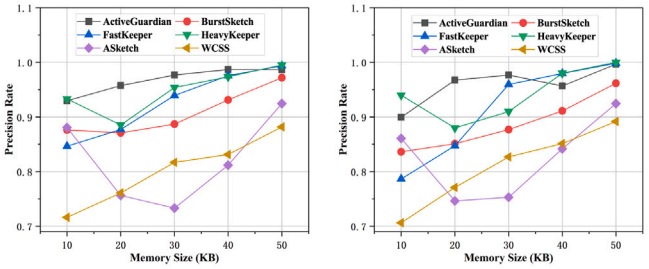
(1) **Precision Rate (PR):** The ratio of the number of correctly reported to the number of reported instances.
(2) **Recall Rate (RR):** The ratio of the number of correctly reported to the number of true instances.
(3) **ARE (Average Relative Error):** $\frac{1}{|Z|} \sum_{f_i \in Z} \frac{|\hat{n}_i - n_i|}{n_i}$, where Z is estimated set of active flows, $\hat{n}_i$ is the estimated size of flow f_i, n_i is the real size of flow f_i. We use ARE to evaluate the accuracy of flow size estimation and the identification of active elephant flows.
(4) **AAE(Absolute Average Error):** AAE is defined as $\frac{1}{|Z|} \sum_{f_i \in Z} |\hat{n}_i - n_i|$, similarly to ARE.
(5) **Throughput:** $\frac{N}{T}$, where N is the total number of packets, and T is the total measurement time. We use Million of insertions per second (Mps) to measure the throughput.

5.3. Parameter settings

In this subsection, we measure the effects of key parameters in ActiveGuardian on its performance, including the filtering threshold α, the guarding threshold β, the update threshold γ, and the window size W. In the following experiments, we set memory to 50 kB and conduct experiments on the MAWI dataset. Fig. 5 illustrates the valuing of performance metrics PR, RR, AAE, and ARE, under different parameter settings.

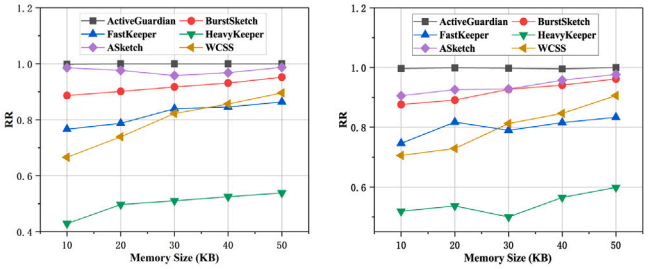
Effects of α: We obtain the performance metrics of ActiveGuardian in Fig. 5(a), by incrementally increasing the filtering threshold α from $2^3 - 1$ to $2^6 - 1$, with the constancy of other parameters. As shown in Fig. 5(a), the optimal value for α is 7 or 15. With the filtering threshold continues to rise, there is a slight increase in RR, while PR decreases, and AAE and ARE increase. This phenomenon is attributed to the fact that the update cycle of the filter becomes longer with a higher filtering threshold. This further lets mice flows easily pass into the Guardian module due to hash collisions, and leads to misjudgments and overestimation of active elephant flows. We also observe that ActiveGuardian achieves superior overall performance, with RR exceeding 99%, PR over 98%, and the lowest AAE and ARE, when α is set to 15. Therefore, α defaults to $2^4 - 1 = 15$.

Effects of β: We incrementally increase the guarding threshold β from 300 to 700, while keeping the remaining parameters constant. With these settings, we gain the performance metrics of ActiveGuardian in Fig. 5(b). We can see from Fig. 5(b) that PR shows an increasing trend, and AAE and ARE gradually decrease with the increase in the guarding threshold. When β = 500, RR exceeds 99%, PR goes beyond 98%, and AAE and ARE reach relatively low levels. As β continues



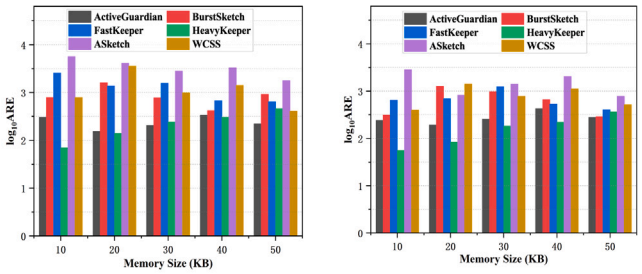
(a) MAWI (b) WebDocs

Fig. 6. Precision vs. memory size.



(a) MAWI (b) WebDocs

Fig. 7. Recall vs. memory size.



(a) MAWI (b) WebDocs

Fig. 8. ARE vs. memory size.

to rise, the performance tends to become stable and is hard to be further optimized. Since our goal is to find out flows that potentially impact network performance, it is unsuitable to set too high guarding threshold. In summary, β is set as 500 by default.

Effects of γ: We achieve the performance metrics of ActiveGuardian in Fig. 5(c), by incrementally increasing the update threshold γ from 0.75 to 0.95, with the constancy of other parameters. As illustrated in Fig. 5(c), RR and AAE show an upward trend while PR exhibits a downward trend, with the growth of the update threshold. This is because higher update threshold means longer update period, which result in that the filter is prone to misjudge mice flows as elephant ones. It is evident that ActiveGuardian achieves optimal overall performance in the case of $\gamma = 0.85$. Therefore, γ defaults to 0.85.

Effects of W: We incrementally increase the window size W from 10K to 200K, while keeping the rest parameters constant. With these settings, we get the performance metrics of ActiveGuardian in Fig. 5(d). We can observe from Fig. 5(d) that, RR keeps above 99%, PR initially rises, then slightly decreases and finally stabilizes at around 98%, while AAE and ARE gradually decrease, for the increasing window size. This suggests that ActiveGuardian consistently maintains excellent performance even as packet traffic grows. In order to report active elephant flows in short periods, W is configured as 100K by default.

5.4. Precision and recall

Precision vs. memory size: As shown in Fig. 6(a) with the MAWI dataset, the precision of ActiveGuardian, BurstSketch, FastKeeper, HeavyKeeper, ASketch, and WCSS is respectively 92.96%, 87.62%, 84.64%, 93.32%, 88.08%, and 71.62% at the memory size of 10 kB. With the increase in memory, the precision of ActiveGuardian gradually increases from approximately 92.96% to around 98.65%. As seen from Fig. 6(b) with the WebDocs dataset, the precision rates for ActiveGuardian, BurstSketch, FastKeeper, HeavyKeeper, ASketch, and WCSS are respectively 90.12%, 83.62%, 78.64%, 94.00%, 86.08%, and 70.62% when the memory size is 10 kB. In addition, ActiveGuardian provides higher accuracy in memory sizes between 20 kB and 50 kB, with the precision rate ranging from approximately 96.76% to 99.65%. In summary, ActiveGuardian achieves relatively high precision rates under the same memory constraints. This is because ActiveGuardian handles mice flows and elephant flows separately for efficient flow tracking, and we can accurately and effectively identify active elephant flows.

Recall vs. memory size: As shown in Fig. 7(a) with the MAWI dataset, the recall rates for ActiveGuardian, BurstSketch, FastKeeper, HeavyKeeper, ASketch, and WCSS are respectively 99.77%, 88.62%, 76.64%, 42.89%, 98.55%, and 66.56% at a memory size of 10 kB. As the memory size increases, the recall rate of ActiveGuardian shows a gradual improvement from approximately 99.77% to around 99.96%. As displayed in Fig. 7(b) with the WebDocs dataset, the recall rates for ActiveGuardian, BurstSketch, FastKeeper, HeavyKeeper, ASketch, and WCSS are respectively 99.67%, 87.62%, 74.64%, 51.89%, 90.55%, and 70.56% when the memory size is 10 kB. ActiveGuardian also maintains relatively high recall rates in memory sizes from 20 kB to 50 kB, ranging from approximately 99.96% to 99.89%. In conclusion, ActiveGuardian consistently achieves high recall rates with the same memory size, and exhibits a pronounced advantage in low-memory scenarios. This indicates that ActiveGuardian is more suitable for resource-constrained measurement environments. This is primarily due to that ActiveGuardian utilizes an *Adaptive Counter Update* strategy to filter out mice flows continuously and effectively even at significantly variable flow rates and with limited memory resources.

5.5. ARE and AAE

ARE vs. memory size: As depicted in Fig. 8(a) with the MAWI dataset, the $\log_{10} ARE$ of ActiveGuardian, BurstSketch, FastKeeper, HeavyKeeper, ASketch, and WCSS are respectively 2.48, 2.90, 3.41, 1.85, 3.75, and 2.90 at a memory size of 10 kB. As the memory size varies, the $\log_{10} ARE$ of ActiveGuardian exhibits relatively low $\log_{10} ARE$ across all memory conditions, ranging from 2.53 to 2.18. Conversely, other solutions like BurstSketch, FastKeeper, ASketch, and WCSS consistently exhibit higher $\log_{10} ARE$, with ASketch reaching a peak $\log_{10} ARE$ of 3.75 at 10 kB memory. As seen from Fig. 8(b) with the WebDocs dataset, the $\log_{10} ARE$ of ActiveGuardian, BurstSketch, FastKeeper, HeavyKeeper, ASketch, and WCSS are respectively 2.38, 2.50, 2.81, 1.75, 3.45, and 2.60 when the memory size is 10 kB. On top of that, ActiveGuardian also maintains relatively low $\log_{10} ARE$ in memory sizes varying from 20 kB to 50 kB, with $\log_{10} ARE$ ranging from approximately 2.96 to 2.71. In brief, ActiveGuardian maintains a relatively low ARE compared with other algorithms. This is primarily because that ActiveGuardian reports active elephant flows and cleans expired information at the end of each time window, significantly enhancing memory utilization.

AAE vs. memory size: As shown in Fig. 9(a) with the MAWI dataset, ActiveGuardian achieves $\log_{10} AAE$ between 2.11 and 2.38, while the $\log_{10} AAE$ of BurstSketch ranges from 2.49 to 2.96, FastKeeper changes from 2.24 to 2.91, HeavyKeeper varies from 2.44 to 2.66, ASketch goes from 2.75 to 2.85, and WCSS varies from 3.15 to

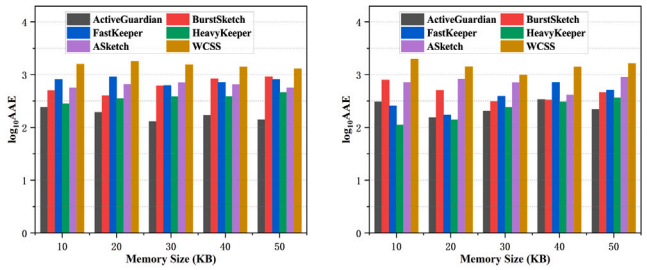


Fig. 9. AAE vs. memory size.

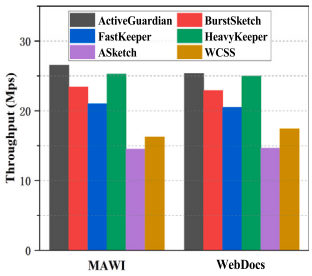


Fig. 10. Throughput of different algorithms.

3.29. As seen from Fig. 9(b) with the WebDocs dataset, ActiveGuardian achieves a $\log_{10} AAE$ between 2.18 and 2.53, while the $\log_{10} AAE$ of BurstSketch changes from 2.49 to 2.96, FastKeeper ranges from 2.24 to 2.85, HeavyKeeper varies from 2.05 to 2.56, ASketch goes from 2.75 to 2.95, and WCSS changes from 2.99 to 3.30. Overall, ActiveGuardian obtains lower AAE than the other algorithms on two real-world datasets, which can further promote the performance of actual network applications. This is because we apply Leapfrog hashing algorithm to provide multiple storage locations for active elephant flows, and timely replace mice flows and low-arrival-rate flows.

5.6. Throughput

Throughput vs. memory size: According to the testbed described in Section 5.1, we allocate 6 CPU cores for packet generation in the sender server, and its throughput can reach 30 Mps. In the receiver server, two dedicated cores are assigned to packet reception, with each core achieving a maximum throughput of 15 Mps. Various flow identification algorithms are deployed on an additional CPU core, tasked with obtaining packet traffic from the receiving cores.

As illustrated in Fig. 10, ActiveGuardian and HeavyKeeper exhibit higher throughput performance compared to ASketch and WCSS for the MAWI dataset. Specifically, ActiveGuardian achieves a throughput of 26.53 Mps and HeavyKeeper achieves 25.27 Mps, while ASketch and WCSS respectively exhibit lower throughput performance, with 14.49 Mps and 16.23 Mps. As for the WebDocs dataset, ActiveGuardian maintains the highest throughput performance of 25.34 Mbps, followed by HeavyKeeper with 24.98 Mps. Conversely, ASketch and WCSS respectively exhibit lower throughput performance, measuring at 14.62 Mbps and 17.43 Mbps. In a nutshell, ActiveGuardian has higher throughput than other algorithms, which can meet the demand of identifying active elephant flows in high-speed network environments. This is mainly owing to that ActiveGuardian utilizes efficient data structures and algorithms, and leverages parallel processing techniques, allowing for simultaneous insertion and query operations.

6. Conclusion

Accurate and efficient identification of active elephant flows is an essential task in network traffic measurement. Due to the unbalanced flow size distribution and changeable flow rates, existing works in real-time measurement suffer from limited precision and high error rates with restricted space memory. In this paper, we propose an algorithm for accurately and efficiently identifying active elephant flows, named ActiveGuardian. We design an *Adaptive Counter Update* strategy to adaptively guarantee the persistent separation of unnecessary flows, and a *Leapfrog Hashing* algorithm to greatly reduce hash collision rates, therefore improving accuracy and efficiency.

Experimental evaluation confirms that ActiveGuardian achieves up to 99.65% precision for identifying active elephant flows, and also maintains around 90% precision even in low-memory scenarios. The recall rate consistently remains above 99%, significantly outperforming existing methods. In summary, ActiveGuardian can achieve accurate identification of active elephant flows with low memory usage. This implies that ActiveGuardian is more suitable for network devices with limited memory resources than other methods. Additionally, ActiveGuardian achieves a higher throughput of 26.53 Mps compared to the state-of-the-art algorithms. This suggests that it has better adaptability to high-speed network environments.

In our future work, we will extend ActiveGuardian to report continuous time windows of active elephant flows. Simultaneously, we will explore robust update strategies and adaptive capacity expansion mechanisms for filtering module and guarding module, for better adaptivity to network traffic jitters or even malicious attacks. In the future, we also plan to deploy ActiveGuardian into various network applications such as intrusion detection systems, network management platforms and online network devices.

CRediT authorship contribution statement

Bing Xiong: Funding acquisition, Methodology, Writing – original draft, Writing – review & editing. **Yongqing Liu:** Formal analysis, Visualization, Writing – review & editing. **Rui Liu:** Formal analysis, Software, Writing – original draft. **Jinyuan Zhao:** Funding acquisition, Investigation, Project administration. **Shiming He:** Funding acquisition, Validation. **Baokang Zhao:** Funding acquisition, Supervision. **Kun Yang:** Conceptualization, Methodology. **Keqin Li:** Conceptualization, Resources.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Shiming He reports financial support was provided by National Natural Science Foundation of China. Baokang Zhao reports financial support was provided by National Natural Science Foundation of China. Bing Xiong reports financial support was provided by Hunan Provincial Natural Science Foundation of China. Bing Xiong reports financial support was provided by Scientific Research Foundation of Hunan Provincial Education Department. Jinyuan Zhao reports financial support was provided by Scientific Research Foundation of Hunan Provincial Education Department. Bing Xiong has patent pending to Changsha University of Science and Technology.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (62272062, U22B2005, 61972412), Hunan Provincial Natural Science Foundation of China (2023JJ30053), and Scientific Research Fund of Hunan Provincial Education Department, PR China (22A0232, 23A0735).

References

Ben-Basat, R., Einziger, G., Friedman, R., et al., 2016. Heavy hitters in streams and sliding windows. In: IEEE International Conference on Computer Communications. INFOCOM, San Francisco, USA, pp. 1–9.

Boyer, R.S., Moore, J.S., 1991. MJRTY—a fast majority vote algorithm. In: Automated Reasoning: Essays in Honor of Woody Bledsoe. Springer Netherlands, Dordrecht, pp. 105–117.

Burnett, S., Chen, L., Creager, D.A., et al., 2020. Network error logging: Client-side measurement of end-to-end web service reliability. In: 17th USENIX Symposium on Networked Systems Design and Implementation. NSDI, Santa Clara, USA, pp. 985–999.

Chang, J.H., Lee, W.S., 2003. Finding recent frequent itemsets adaptively over online data streams. In: 9th ACM Conference on Knowledge Discovery and Data Mining. SIGKDD, Washington, USA, pp. 487–492.

Charikar, M., Chen, K., Farach-Colton, M., 2004. Finding frequent items in data streams. Theoret. Comput. Sci. 312 (1), 3–15.

Chen, P., Chen, D., Zheng, L., et al., 2021. Out of many we are one: Measuring item batch with clock-sketch. In: ACM Special Interest Group on Management of Data. SIGMOD, Xi'an, China, pp. 261–273.

Cheung, Y.L., Fu, A.W.C., 2004. Mining frequent itemsets without support threshold: with and without item constraints. IEEE Trans. Knowl. Data Eng. 16 (9), 1052–1069.

Cormode, G., Muthukrishnan, S., 2005. An improved data stream summary: the count-min sketch and its applications. J. Algorithms 55 (1), 58–75.

Curtis, A.R., Mogul, J.C., Tourrilhes, J., et al., 2011. DevoFlow: Scaling flow management for high-performance networks. In: ACM Special Interest Group on Data Communication. SIGCOMM, Toronto, Canada, pp. 254–265.

Real-life transactional dataset. 2005, http://fimi.ua.ac.be/data/.

Davenport, M., 2012. Introduction to modern information retrieval. J. Med. Libr. Assoc.: JMLA 100 (1), 75.

Deng, F., Rafiei, D., 2007. New estimation algorithms for streaming data: Count-min can do more. Webdocs. Cs. Ualberta. Ca.

Estan, C., Varghese, G., 2003. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. ACM Trans. Comput. Syst. (TOCS) 21 (3), 270–313.

Feldmann, A., Greenberg, A., Lund, C., et al., 2001. Deriving traffic demands for operational IP networks: Methodology and experience. IEEE/ACM Trans. Netw. 9 (3), 265–279.

Garcia-Teodoro, P., Diaz-Verdejo, J., Maciá-Fernández, G., et al., 2009. Anomaly-based network intrusion detection: Techniques, systems and challenges. Comput. Secur. 28 (1–2), 18–28.

Gou, X., He, L., Zhang, Y., et al., 2020. Sliding sketches: A framework using time zones for data stream processing in sliding windows. In: 26th ACM Conference on Knowledge Discovery and Data Mining. SIGKDD, New York, USA, pp. 1015–1025.

Liao, L.X., Chao, H.C., Chen, M.Y., 2020. Intelligently modeling, detecting, and scheduling elephant flows in software defined energy cloud: A survey. J. Parallel Distrib. Comput. 146, 64–78.

MAWILab, 2019. Mawi working group traffic archive. http://mawi.wide.ad.jp/mawi/.

Metwally, A., Agrawal, D., Abbadi, A.E., 2005. Efficient computation of frequent and top-k elements in data streams. In: International Conference on Database Theory. Berlin, Heidelberg, pp. 398–412.

Mirylenka, K., Cormode, G., Palpanas, T., et al., 2015. Conditional heavy hitters: detecting interesting correlations in data streams. VLDB J. 24, 395–414.

Roy, P., Khan, A., Alonso, G., 2016. Augmented sketch: Faster and more accurate stream processing. In: ACM Special Interest Group on Management of Data. SIGMOD, California, USA, pp. 1449–1463.

Sarrar, N., Uhlig, S., Feldmann, A., et al., 2012. Leveraging Zipf's law for traffic offloading. ACM SIGCOMM Comput. Commun. Rev. 42 (1), 16–22.

Sivaraman, A., Subramanian, S., Alizadeh, M., et al., 2016. Programmable packet scheduling at line rate. In: ACM Special Interest Group on Data Communication. SIGCOMM, Florianopolis, Brazil, pp. 44–57.

Tang, L., Huang, Q., Lee, P.P.C., 2019. Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. In: IEEE International Conference on Computer Communications. INFOCOM, Paris, France, pp. 2026–2034.

Wang, Y., Li, D., Wu, J., 2021. FastKeeper: A fast algorithm for identifying top-k real-time large flows. In: IEEE Global Communications Conference. GLOBECOM, Madrid Madrid, Spain, pp. 01–07.

Wang, W., Zhang, X., Shi, W., et al., 2011. Network traffic monitoring, analysis and anomaly detection. IEEE Netw. 25 (3), 6–7.

Yang, T., Gong, J., Zhang, H., et al., 2018a. HeavyGuardian: Separate and guard hot items in data streams. In: 24th ACM International Conference on Knowledge Discovery and Data Mining. SIGKDD, New York, USA, pp. 2584–2593.

Yang, T., Jiang, J., Liu, P., et al., 2018b. Elastic sketch: Adaptive and fast network-wide measurements. In: ACM Special Interest Group on Data Communication. SIGCOMM, Budapest, Hungary, pp. 561–575.

Yang, T., Zhang, H., Li, J., et al., 2019. HeavyKeeper: An accurate algorithm for finding top-k elephant flows. IEEE/ACM Trans. Netw. 27 (5), 1845–1858.

Yang, T., Zhou, Y., Jin, H., et al., 2017. Pyramid sketch: A sketch framework for frequency estimation of data streams. Proc. VLDB Endow. 10 (11), 1442–1453.

Yu, X., Xu, H., Yao, D., et al., 2018. CountMax: A lightweight and cooperative sketch measurement for software-defined networks. IEEE/ACM Trans. Netw. (ToN) 26 (6), 2774–2786.

Zhong, Z., Yan, S., Li, Z., et al., 2021. Burstsketch: Finding bursts in data streams. In: ACM Special Interest Group on Management of Data. SIGMOD, Xi'an, China, pp. 2375–2383.

Zhou, Y., Yang, T., Jiang, J., et al., 2018. Cold filter: A meta-framework for faster and more accurate stream processing. In: ACM Special Interest Group on Management of Data. SIGMOD, New York, USA, pp. 741–756.

Zhu, J., Jiang, X., Yu, Y., et al., 2020. An efficient priority-driven congestion control algorithm for data center networks. China Commun. 17 (6), 37–50.

Zhuang, W., Shen, Y., Li, L., et al., 2021. Develop an adaptive real-time indoor intrusion detection system based on empirical analysis of OFDM subcarriers. Sensors 21 (7), 2287.

Bing Xiong received the Ph.D. degree in Computer Science by master-doctorate program from Huazhong University of Science and Technology (HUST), China, in 2009, and the B.S. degree from Hubei Normal University, China, in 2004. He worked as a visiting scholar in the Department of Computer and Information Science, Temple University, USA, from 2018 to 2019. He is currently an associate professor in the School of Computer and Communication Engineering, Changsha University of Science and Technology, China. His main research interests include furture network architecture, network measurements, network security and artificial intelligence applications..

Yongqing Liu received the B.S. degree in Network Engineering from Changsha University of Science and Technology, China, in 2019. He is currently pursuing the M.S. degree with Changsha University of Science and Technology, advised by B. Xiong. His research interests include network measurements, sketches and data stream processing.

Rui Liu received the B.S. degree in Network Engineering from Jiangxi University of Technology, China, in 2020. He is currently pursuing the M.S. degree with Changsha University of Science and Technology, advised by B. Xiong. His research interests include network measurements, sketches and data stream processing.

Jinyuan Zhao received the Ph.D. degree in Computer Science from Central South University, China, in 2020, and the M.S. degree from Central China Normal University, China, in 2007. She worked in the School of Computer and Communication, Hunan Institute of Engineering, China, from 2007 to 2020. She is currently an assistant professor in the School of Information Science and Engineering, Changsha Normal University, China. Her main research interests include future network architecture, network measurements.

Shiming He received the B.S. degree in information security and a Ph.D. degree in computer science and technology from Hunan University, China, in 2006 and 2013, respectively. She is currently an Associated Professor with the School of Computer and Communication Engineering, Changsha University of Science and Technology, Changsha, China. Her research interests include machine learning, data analysis, and anomaly detection.

Baokang Zhao received the B.S., M.S., and Ph.D. degrees from National University of Defense Technology, all in computer science. He is currently an Associate Professor in the School of Computer Science, NUDT. His research interests include system design, protocols, algorithms, and security issues in computer networks.

Kun Yang received his Ph.D. from the Department of Electronic & Electrical Engineering of University College London (UCL), UK. He is currently a Chair Professor in the School of Computer Science & Electronic Engineering, University of Essex, leading the Network Convergence Laboratory (NCL), UK. He is also an affiliated professor at UESTC, China. Before joining in the University of Essex at 2003, he worked at UCL on several European Union (EU) research projects for several years. His main research interests include wireless networks and communications, IoT networking, data and energy integrated networks and mobile computing. He manages research projects funded by various sources such as UK EPSRC, EU FP7/H2020 and industries. He has published 400+ papers and filed 30 patents. He serves on the editorial boards of both IEEE (e.g., IEEE TNSE, IEEE ComMag, IEEE WCL) and non-IEEE journals (e.g., Deputy EiC of IET Smart Cities). He was an IEEE ComSoc Distinguished Lecturer (2020–2021). He is a Member of Academia Europaea (MAE), a Fellow of IEEE, a Fellow of IET and a Distinguished Member of ACM.

Keqin Li received a B.S. degree in computer science from Tsinghua University in 1985 and a Ph.D. degree in computer science from the University of Houston in 1990. He is a SUNY Distinguished Professor with the State University of New York and a National Distinguished Professor with Hunan University (China). He has authored or co-authored

more than 970 journal articles, book chapters, and refereed conference papers. He received several best paper awards from international conferences including PDPTA-1996, NAECON-1997, IPDPS-2000, ISPA-2016, NPC-2019, ISPA-2019, and CPSCom-2022. He holds nearly 75 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top five most influential scientists in parallel and distributed computing in terms of single-year and career-long impacts based on a composite indicator of the Scopus citation database. He was a 2017 recipient of the Albert Nelson Marquis Lifetime Achievement Award for being listed in Marquis Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, and Who's Who in American Education for over twenty consecutive years. He received the Distinguished Alumnus Award from the Computer Science Department at the University of Houston in 2018. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He is a Member of the SUNY Distinguished Academy. He is an AAAS Fellow, an IEEE Fellow, an AAIA Fellow, and an ACIS Founding Fellow. He is an Academician Member of the International Artificial Intelligence Industry Alliance. He is a Member of Academia Europaea (Academician of the Academy of Europe).