OXFORD

The Computer Journal, 2024, **67**, 2039–2053
https://doi.org/10.1093/comjnl/bxad122
Advance access publication date 20 December 2023
Original Article

A Sketch Framework for Fast, Accurate and Fine-Grained Analysis of Application Traffic

Changsheng Hou, Chunbo Jia, Bingnan Hou*, Tongqing Zhou, Yingwen Chen and Zhiping Cai*

College of Computer, National University of Defense Technology, Changsha 410073, China

*Corresponding author: houbingnan19@nudt.edu.cn, zpcai@nudt.edu.cn

Nowadays, with the continuous increase in internet traffic, the demand for real-time and high-speed traffic analysis has grown significantly. However, existing traffic analysis technologies are either limited by specific applications or data, unable to expand for widespread implementation, or in offline mode are unable to keep up with dynamic adjustments required in certain network management scenarios. A promising approach is to utilize sketch technology to enhance real-time traffic analysis. Unfortunately, existing technologies suffer from defects, such as overly coarse-grained statistics that cannot perform precise application-level traffic analysis, and irreversibility, which cannot support real-time queries in a friendly way. To achieve real-time fine-grained application traffic analysis in general scenarios, we propose AppSketch, a real-time network traffic measurement tool. AppSketch adopts a one-pass approach to classify and label the application information of each packet in the network flows. It then hashes the flow, identified with the application tag, into a carefully designed multiple-key sketch, for gathering application-specific statistics. We conducted extensive experiments using a real-world network traffic dataset collected on a university campus. The results showed that AppSketch achieved high accuracy while requiring less update time than other alternatives. Moreover, AppSketch occupies limited memory (≤64KB), making it suitable for online network devices.

Keywords: network measurement; application traffic analysis; sketch

1. INTRODUCTION

The internet carries a variety of traffic at all times, including financial transactions, e-commerce, entertainment and more [1–3]. Due to the massive volume of network traffic, it has been reported that each link in large internet service providers (ISPs) or data centers handles millions of packets per second [4–7]. Real-time and high-speed traffic analysis, such as identifying traffic trends, application proportions and flow sizes, is critical for ISPs or companies to optimize their network management and ensure high-quality service (QoS). Moreover, network attacks are often hidden in massive legitimate and benign traffic to evade detection by defense mechanisms [8]. Despite the continuous improvement in accuracy and efficiency of existing attack traffic detection methods [8, 9], due to the lack of fine-grained analysis for traffic, they are still unable to cope with the emerging adversarial attack traffic generation technologies [10–12]. Multidimensional and fine-grained traffic analysis can provide insights for more refined traffic attack detection.

The existing traffic analysis techniques can be primarily classified into two modes: offline and online. Developed traffic analysis technologies, such as those based on Deep Packet Inspection (DPI) and machine learning, are almost offline. DPI is a traffic monitoring and analysis technology that involves inspecting all packet payloads to match application signatures, resulting in high computational overhead. Given the continuous improvement of the packet processing capability of network equipment, it becomes increasingly challenging for DPI-based technology to match the line rate [13]. Most machine learning-based methods require training the model offline, which leads to the inability to perform real-time analysis on the traffic [14]. Some rapid machine learning classification techniques provide near real-time traffic analysis capabilities [15, 16]; however, the study focuses on specific applications and may not be suitable for effectively handling the broader spectrum of application traffic. For online traffic analysis, researchers realize real-time network traffic monitoring through software-defined networking (SDN) and network function virtualization (NFV) technologies [17] or realize online flow calculation through hardware acceleration [18]. However, these methods still have some concerns. For example, network traffic monitoring collects comprehensive traffic data, which poses privacy risks; is only applicable to SDN scenarios and cannot be deployed in non-SDN networks; and hardware acceleration is customized and can only be used for specific data, such as dense (non-sparse) correlation matrices, and thus is not scalable.

Real-time traffic analysis is essential in order to allocate network resources efficiently and provide per-flow QoS [15, 19]. One natural approach is to use sketch technology for real-time traffic statistics. However, there are some challenges in integrating sketch technology into application traffic analysis.

One challenge is that existing sketch-based algorithms are still too coarse-grained to provide detailed statistics on applications and sources [20]. While related works, such as FlexSketchMon [21], focus on implementing generalized traffic estimation techniques like superspreader detection and heavy hitter detection, they often lack the ability to enable more precise application-level traffic analysis. *Another challenge is reversibility, as traffic analysis from the application point of view requires sketches to be reversible.* Classic sketches such as Count-min Sketch [22] are irreversible and cannot support real-time queries in a friendly way. This makes

Received: April 12, 2023. **Revised:** October 14, 2023. **Accepted:** November 28, 2023

it difficult to recover all heavy flows from only the sketch data structure itself, and instead, a computationally expensive brute-force approach must be used to query every possible flow to check whether it is a heavy flow.

In this work, we propose a novel sketch framework called AppSketch, designed for application traffic analysis. The framework utilizes lightweight classification of traffic and sketch techniques to enable real-time and fine-grained traffic analysis, including identifying top-k traffic consumption apps and top-k source IP of traffic requests. Firstly, AppSketch quickly classifies network traffic according to the application type, assigning each packet in the online stream with an application tag. It then employs a novel multiple-key sketch to enable efficient full-key and arbitrary partial-key queries. By specifying a full key (incorporating all partial keys that might be queried in the future), AppSketch supports querying for aggregate flows defined by both full and partial keys. For example, by specifying a tuple consisting of an application tag and a source IP as the full key, AppSketch can perform top-k flows, top-k applications and top-k source IP queries. To achieve reversibility and perform partial key queries efficiently, AppSketch records a full key within the bucket and updates the recorded key using the random variance minimization technique. In summary, our work makes the following contributions:

1. Our work proposes a novel sketch framework, named AppSketch, which combines lightweight traffic classification with sketch techniques to enable fast and accurate traffic statistics and analysis at a fine-grained level.
2. To evaluate the performance of AppSketch, we conducted experiments on real-world IP-trace streams. The results demonstrate the superior accuracy and efficiency of AppSketch compared with existing methods. We also applied AppSketch to analyze the main applications and sources in a real campus network from China. The findings reveal that HTTP/HTTPS and audio/video applications account for the majority of traffic, and most heavy sources interact with data centers, possibly involving the acquisition of experimental data.

The rest of the paper is organized as follows: Section 2 discusses the background and related work. Section 3 describes the design of AppSketch. Section 4 presents the flow statistics and related query operations. Section 5 evaluates the performance of AppSketch. We show a use case in Section 6 and Section 7 concludes this work.

2. BACKGROUND AND RELATED WORK

2.1. Lightweight Packet Inspection

Currently, traffic classification techniques can be broadly classified into three categories: port-based, machine learning-based [23] and payload-based techniques. The port-based approach is widely considered inadequate since many applications do not use a well-defined port [24]. While machine learning-based methods generally offer high accuracy, they often rely on offline training, which cannot meet the real-time demands of traffic classification. To overcome this limitation, researchers have explored the use of rapid machine learning classification techniques, such as the C4.5 Decision Tree [15]. These techniques provide near real-time traffic analysis capabilities. However, it is important to note that the study [15] focuses on classifying specific applications such as online games and VoIP applications, and may not be suitable for effectively handling the broader spectrum of application traffic. Consequently, most traffic classifiers rely on payload-based approaches using DPI (e.g. OpenDPI [25] and L7-Filter [26]) or Layer 7 Protocol Identification (LPI) (e.g. Libprotoident [24]). The DPI approach can achieve accurate traffic classification, but it suffers from high computational overhead and poses an increased risk of user privacy disclosure, as it requires access to the entire payload of each packet.

Instead of analyzing the entire payload, the LPI approach reduces the inspection of packet content to improve efficiency while maintaining relatively high classification accuracy. Among the LPI approaches, the Libprotoident [24] library achieves even higher accuracy than DPI approaches, as demonstrated in several previous studies [27, 28]. The Libprotoident library identifies the application protocol by analyzing the first four bytes of payload from a few packets, the size of the first packet sent in each direction and the port numbers used by each endpoint [29]. In our work, we utilize the Libprotoident library for traffic classification.

2.2. Sketch-based Approximate Algorithms

Sketches are stream data aggregation structures that track values in a fixed number of buckets. Classical sketches (e.g. Count Sketch [30], K-ary Sketch [31] and Count-min Sketch [22]) linearly project stream data into spaces with lower dimensions yet maintain the aggregation characteristics of the data. Take Count-min Sketch [22] as an example, which constructs the sketch as w rows and h buckets in each row. Each bucket is associated with a counter initialized as 0. At the timestamp t, for the incoming key-value pairs, e.g. $(x, f; t)$, it hashes x into a bucket in each of the w rows using w pairwise independent hash functions. Then it increases the counter in each of the w buckets by f. Since hash collisions will result in multiple keys being projected into the same bucket. To this end, Count-min Sketch uses the minimum value in the buckets mapping to key x as the estimated value.

Several recent studies have focused on improving the sketch structure to address issues such as low packet processing rates and poor flow estimation accuracy. For example, SketchVisor [32] redirects high traffic loads that traditional sketch solutions cannot handle to a separate fast path, enabling fast but slightly less accurate measurements. Elastic Sketch [33] proposes techniques for compressing/merging sketches to improve their adaptability to bandwidth, packet rate and flow size distribution. NitroSketch [34] reduces per-packet CPU and memory operations through sampling on counter arrays, adaptive sampling and other methods. PR-Sketch [35] divides flow tracking into update and recovery phases to achieve distributed updates and centralized recovery. Other approaches, such as SeqSketch, EmbedSketch [36] and LightGuardian [37], have also been proposed. However, our work focuses on real-time and partial-key query features for fine-grained traffic analysis.

In recent years, there have been various efforts to address the frequent items (or hot items) query problem and to implement reversible sketches. For example, Cold Filter (CF) [38] proposes a meta-framework that captures cold items in the first stage and hot items in the second stage, which can be combined with Space-Saving [39] to record frequent items. HeavyGuardian [40] divides each bucket into a heavy part and a light part and uses exponential decay technology to separate hot items from cold items. WavingSketch [41] is an unbiased estimation algorithm for finding frequent items that divides the bucket into a counter and a heavy part, and keeps track of frequent items by continuously updating the heavy part list.

In addition to efforts aimed at single-key queries, Unbiased Space Saving (USS) [42] provides solutions to the arbitrary partial-key query problem. Unfortunately, the update latency of

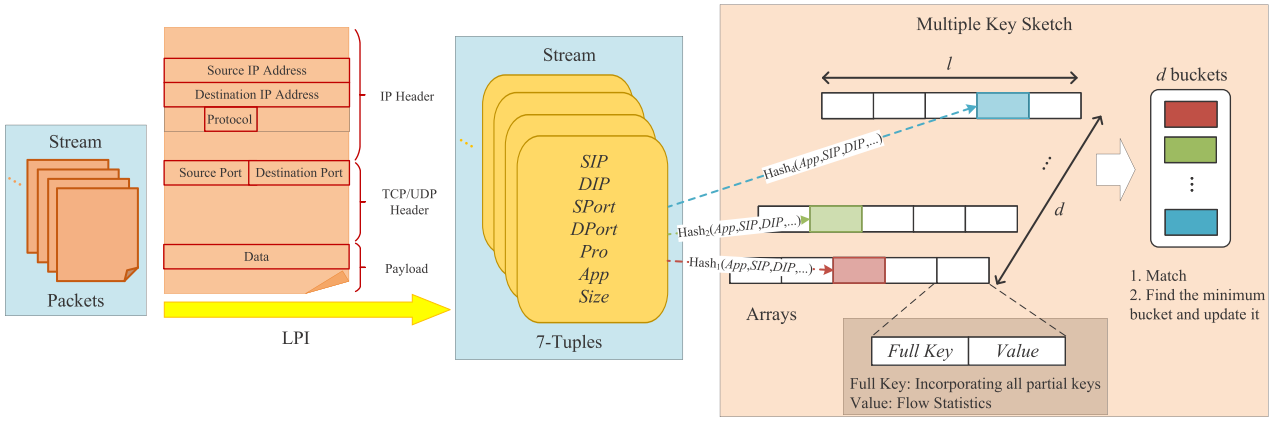

Figure 1. The framework of AppSketch.

USS grows proportionally with the number of recorded flows. CocoSketch [43] proposes random variance minimization to reduce the per-packet update delay. DMatrix [44] is designed to be three-dimensional to capture flow information from multiple dimensions, while a field is reserved in each bucket for the key of interest, reducing the computational cost of key recovery with an acceptable memory overhead.

Furthermore, the Chinese Remainder Theorem based Reversible Sketch (CRT-RS) [45] and ExtendedSketch [46] implement reversible queries based on the Chinese remainder theorem. However, the effectiveness of these two methods is highly dependent on the correct configuration of parameters, which limits their applicability and scalability.

3. APPSKETCH DESIGN

AppSketch is a sketch framework that supports lightweight and real-time network traffic analysis. The main designing goals of AppSketch are as follows: (1) **Rapid traffic classification**: AppSketch adopts lightweight packet inspection (LPI) technology to implement online traffic classification to support fine-grained traffic analysis. (2) **Real-time traffic analysis**: AppSketch is designed to be reversible, different from the traversal and module hash techniques; it can readily return all heavy keys (e.g. the top-k traffic consumption applications) from the sketch structure itself, thereby improving the real-time performance of traffic analysis. (3) **Multiple key queries**: AppSketch supports queries for multiple keys without the need to pre-define which keys need to be measured.

3.1. Overview on AppSketch

AppSketch is a two-stage framework for application traffic analysis shown in Fig. 1. In the first stage, we adopt LPI techniques to quickly classify online flows, assigning each packet an application tag and its packet size. In the first stage, application-layer network traffic classification is performed on the network flow. For bidirectional flow, LPI extracts some fields of IP header and transport layer header, and a small amount of payload, to grasp the information of flows in terms of port, payload, statistical characteristics and support network traffic classification. LPI labels each packet with the application type and packet size. Combined with the directly obtained 5-tuple, AppSketch forms a 7-tuple for each packet (i.e. source IP, destination IP, source port, destination port, protocol, application and packet size). In the second stage, each 7-tuple is hashed into a multiple-key sketch. Multiple-key sketch supports queries for both full key and arbitrary partial keys. By specifying only a full key (incorporating all partial keys that might be queried in the future), sketch supports queries for aggregate flows defined by arbitrary keys. Here, a full key contain all the elements that define the flow, while a partial key only contain part of elements. For example, if we specify a 5-tuple <source IP, destination IP, source port, destination port, protocol> as a full key, then <source IP, source port> and <destination IP, destination port> are both partial keys.

Figure 1 shows the structure of AppSketch which is composed of d bucket arrays, each containing l buckets, namely a total of $d \times l$ buckets. The hash function $Hash(\cdot)$ defines a mapping of a full key (e.g. $\langle App, SIP, DIP \rangle$) from its key space to a hash value (e.g. $Hash(\langle App, SIP, DIP \rangle)$). A full key can be any combination of elements that can define a flow (e.g. 5-tuple, application tag, etc.). Since the definition of the full key can be arbitrary and depends on different analysis objectives, in the following elaboration of this paper, $\langle App, SIP \rangle$ is used as the full key to illustrate the operations. The operations of other defined full keys are similar. In the second stage, AppSketch uses the d hash functions to select a bucket in each bucket array to form a list of d buckets. Then, AppSketch performs the update process.

3.2. Sketch Update

To realize the reversibility of AppSketch, each bucket is divided into two fields. The first field *Full Key* records the candidate key of possible heavy flow, and the second field *Value* records the statistics (e.g. packet count or byte count) of the flow. For more granular traffic analysis, AppSketch is designed to better support partial key queries (e.g. queries for top-k traffic consumption apps, top-k sources of traffic requests, etc.).

AppSketch utilizes the stochastic variance minimization technique in CocoSketch [43] to achieve more accurate partial key queries. Specifically, AppSketch uses d hash functions to select a bucket in each array to form a list containing d buckets. Then, AppSketch traverses this list and checks whether the key of the incoming flow matches the *Full Key* field of a bucket. If the key of the incoming flow matches the *Full Key* field in one of the d buckets, the *Value* field of the corresponding bucket is updated. If the incoming flow does not match any of the records in the d buckets, AppSketch selects the bucket with the *Value* field being the minimum of the d buckets and updates it. The value of the incoming flow is accumulated to the minimum bucket, and the *Full Key* field of that bucket is replaced with the key of the incoming flow according to a probability.

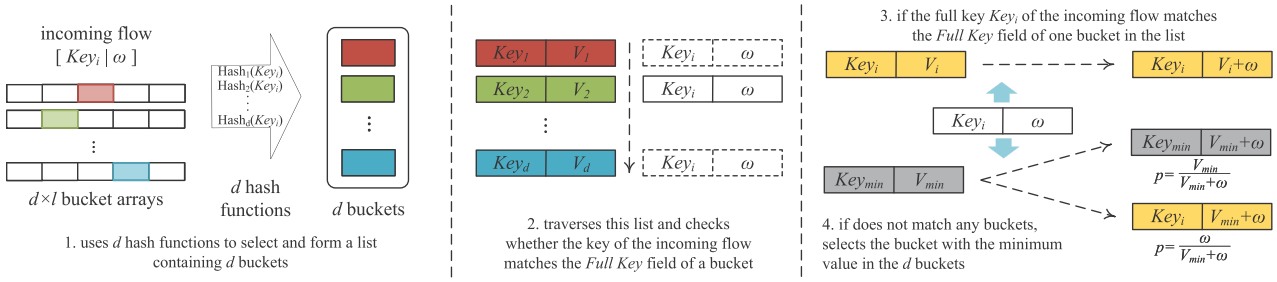

Figure 2. Sketch Update Process.

Assume that the full key of the incoming flow is $Key_i = (\langle App_i, SIP_i \rangle)$, and the corresponding value is ω. Figure 2 shows the detailed process of updating sketch. First, AppSketch uses a set of d hash functions and the key Key_i of the incoming flow to form a bucket list from the $d \times l$ bucket arrays. Then, AppSketch traverses this list and compares the incoming flow with the records in the bucket list one by one. If Key_i matches the *Full Key* field in the d buckets (suppose the value in this bucket is V_i), the *Value* field of the bucket is updated to $V_i + \omega$. If it does not match, AppSketch selects the bucket with the minimum value in the d buckets, and the full key and value of the minimum bucket are Key_{min} and V_{min}, respectively. Then, the *Value* field is updated to $V_{min} + \omega$, and the *Full Key* field is replaced with Key_i with probability $\omega/(V_{min} + \omega)$.

Unlike single-key sketch which seeks to minimize the maximum estimation error on individual keys, AppSketch aims to minimize the estimation error of a subset sum, i.e. minimize $\sum_{Key} \left(V(Key) - \hat{V}(Key)\right)^2$. Since a partial key query is actually querying for an aggregated result of a certain subset of all flows, minimizing the subset sum estimation error makes AppSketch guarantee the accuracy of partial key queries.

Algorithm 1. Sketch Update

Input: Stream S
Output: Updated AppSketch B

1: **for** $s(SIP, DIP, SPort, DPort, Pro, App, \omega) \in$ Stream S **do**
2: $Value_{min} = MAX, i_{min} = 0, j_{min} = 0$
3: **for** $i = 1$ to d **do**
4: $j = Hash_i(\langle App, SIP \rangle)$
5: **if** $B(i,j).Full\_Key == \langle App, SIP \rangle$ **then**
6: $B(i,j).Value += \omega$
7: **return**
8: **else**
9: **if** $B(i,j).Value < Value_{min}$ **then**
10: $Value_{min} = B(k,i).Value$
11: $i_{min} = i$
12: $j_{min} = j$
13: **end if**
14: **end if**
15: **end for**
16: $B(i_{min}, j_{min}).Value += \omega$
17: **if** $random()\%B(i_{min}, j_{min}).Value < \omega$ **then**
18: $B(i,j).Full\_Key = \langle App, SIP \rangle$
19: **end if**
20: **end for**

As shown in Algorithm 1, the process of updating AppSketch is fairly straightforward. For each incoming tuple (abbreviated as $s(SIP, DIP, SPort, DPort, Pro, App, \omega)$), we first initialize $Value_{min}$, i_{min} and j_{min} (Line 2). Next, we compute the index of its mapping bucket $B(i,j)$ where $j = Hash_i(\langle App, SIP \rangle)$, $i \in [1, d]$ (Line 4). Then we check if $\langle App, SIP \rangle$ matches the *Full Key* field of bucket $B(i,j)$. If $B(i,j).Full\_Key == \langle App, SIP \rangle$, we increase the *Value* field of $B(i,j)$ by ω (Lines 5–7). Otherwise, we check if $B(i,j)$ is the bucket with a smaller value, and if so, we record the minimum value and the coordinates of the minimum bucket (Lines 8–13). After traversing d buckets, if $\langle App, SIP \rangle$ does not match the *Full Key* field of any bucket, then we update the minimum bucket. We increase ω to the *Value* field of the minimum bucket, and then, we obtain a random number by *random*() and update the *Full Key* field of the minimum bucket to $\langle App, SIP \rangle$ with probability $\omega/B(i_{min}, j_{min}).Value$ (Lines 16–19).

4. FLOW STATISTICS AND QUERY OPERATIONS

AppSketch supports full key query and arbitrary partial key query, mainly including: (1) the requested network flow, (2) the top-k heavy-flows, (3) the top-k heavy-applications and (4) the top-k heavy-sources. We use heavy-applications (resp. heavy-sources) here refer to heavy-hitter on apps (resp. IP addresses) in the stream data. In the following text, we will describe the algorithms for calculating these statistics in more detail.

Algorithm 2. Query for Network Flow

Input: Querying flow $\langle App, SIP \rangle$
Output: Size estimation $\tilde{V}(\langle App, SIP \rangle)$ of flow $\langle App, SIP \rangle$

1: **for** $i = 1$ to d **do**
2: $j = Hash_i(\langle App, SIP \rangle)$
3: **if** $B(i,j).Full\_Key == \langle App, SIP \rangle$ **then**
4: **return** $B(i,j).Value$
5: **end if**
6: **end for**
7: **return** 0

4.1. Network Flow Query

The query operation on the size of a certain flow is shown in Algorithm 2. For a flow determined by the application tag *App* and the source IP address *SIP* in the network, we calculate the estimated value $\tilde{V}(\langle App, SIP \rangle)$ of the flow size $V(\langle App, SIP \rangle)$. According to the full key $\langle App, SIP \rangle$ of the queried flow, in each bucket array, AppSketch checks the *Full Key* field of the bucket determined by the hash function $Hash_i(\cdot)$ (Line 3). If it matches, the value of the *Value* field of that bucket is returned (Line 4). After traversing d buckets, if $\langle App, SIP \rangle$ does not match the *Full Key* field of any bucket, an estimate of **0** is returned (Line 7).

Here, AppSketch achieves unbiased estimation, which can be elucidated by analyzing the update sketch process. Supposing

(Key_i, w) is the t-th incoming packet, let $\widehat{f}^t(x)$ denote the estimated size of flow x before the t-th insertion. We demonstrate unbiasedness by showing that for records in buckets, the expected increment is w if $x == Key_i$, and 0 otherwise. For the case $x == Key_i$, if x has been recorded, the estimated size increases accordingly by w. If x is not recorded, an increase in the estimated size of flow x only occurs when the *Full Key* field of the minimum bucket is replaced, with an expected increment of $(V_{min} + w) \times w/(V_{min} + w) = w$. For the case $x \neq Key_i$, the estimated size of flow x changes unawares only when x has been recorded in the minimum bucket among the bucket list, with an expected increment of $(\widehat{f}^t(x) + w) \times \widehat{f}^t(x)/(\widehat{f}^t(x) + w) - \widehat{f}^t(x) = 0$. Therefore, the query results of flow x is an unbiased estimation. Algorithm 2 reveals that AppSketch underestimates the value of flows that are not in the sketch to 0. Since AppSketch achieves unbiased estimation using the stochastic variance minimization technique, AppSketch overestimates the flows that are in the sketch.

We roughly estimate the time complexity of the query operation by counting the number of bucket accesses. Clearly, the time complexity to estimate the weight of a certain flow is $\Theta(d)$. Note that the time complexity here refers to querying only one flow. When performing multiple queries (or querying multiple flows), consider the number of queries N_{query}, i.e. $\Theta(d \cdot N_{query})$.

Algorithm 3. Query for Top-k Heavy-flows

Input: Querying number K
Output: Top-k heavy-flows $\mathcal{L}_{flow}$
1: $\mathcal{L}_{flow} = 0$
2: **for** $i = 1$ to d **do**
3: **for** $j = 1$ to l **do**
4: $\mathcal{L}_{flow} \leftarrow (B(i,j).Full\_Key, B(i,j).Value)$
5: **end for**
6: **end for**
7: $ReorderAndTruncate(\mathcal{L}_{flow})$
8: **return** $\mathcal{L}_{flow}$

4.2. Top-k Heavy-flows

Algorithm 3 shows the query operation on top-k heavy-flows. We first initialize a list $\mathcal{L}_{flow}$ (Line 1). Then we traverse the entire bucket arrays, inserting the full keys and values stored in each bucket into the list $\mathcal{L}_{flow}$ (Lines 2–6). Finally, the list $\mathcal{L}_{flow}$ is reordered and truncated (or complemented) in descending order of the values (Line 7). Note that the truncation or complementation here is based on the relationship between the number of buckets $d \times l$ and the requested K. If $d \times l < K$, since AppSketch cannot return a sufficient number of heavy flows, the list $\mathcal{L}_{flow}$ is complemented with the default value $(Null, 0)$. Otherwise, the list $\mathcal{L}_{flow}$ is truncated to be of length K.

The time complexity of traversing the bucket array is $\Theta(dl)$. Then, we consider the process of reordering and truncating, which uses the Quicksort algorithm (the time complexity of sorting a list with n elements is $O(n \cdot log(n))$). After traversing the bucket array, the list $\mathcal{L}_{flow}$ contains $d \times l$ elements. Therefore, the time complexity of the reordering and the truncating process is $O(dl \cdot log(dl))$. Therefore, the time complexity of querying the top-k flows is $\Theta(dl \cdot log(dl))$.

4.3. Top-k Heavy-applications

Top-k heavy-applications mean the top-k applications that consume the most traffic among the active applications on the network. Video applications are usually heavy applications, since the video files are large and consume a lot of network bandwidth resources during transmission. In addition, P2P applications and file-sharing applications are frequently used by people, and therefore generate a lot of traffic. Furthermore, voice communication applications and life assistance software unconsciously consume a lot of traffic.

Algorithm 4. Query for Top-k Heavy-applications

Input: Querying number K
Output: Top-k heavy-applications $\mathcal{L}_{app}$
1: $\mathcal{L}_{app} = 0$
2: **for** $i = 1$ to d **do**
3: **for** $j = 1$ to l **do**
4: $\mathcal{L}_{app} \leftarrow (B(i,j).Full\_Key.App, B(i,j).Value)$
5: **end for**
6: **end for**
7: $ReorderAndTruncate(\mathcal{L}_{app})$
8: **return** $\mathcal{L}_{app}$

Algorithm 4 shows the query operation on top-k heavy-applications. The query for top-k heavy-applications is similar to the query for top-k heavy-flows. After initializing list $\mathcal{L}_{app}$, we traverse the entire bucket arrays and insert the corresponding key-value pairs into list $\mathcal{L}_{app}$. Note that here we extract the corresponding partial key, i.e. *App*, and insert the partial key and value into list $\mathcal{L}_{app}$. When inserting, if the applications tag is already recorded in list $\mathcal{L}_{app}$, it is accumulated over the original value. Otherwise, the new key-value pair $(B(\cdot).Full\_Key.App, B(\cdot).Value)$ is inserted.

In the most extreme case, list $\mathcal{L}_{app}$ contains dl elements. Based on this, we roughly estimate the time complexity of top-k application query. When traversing the bucket array, the key-value pair in each bucket is accumulated to the appropriate position in the list $\mathcal{L}_{app}$. Therefore, the time complexity of key matching (i.e. the process compared with the dl elements in the list one by one) is $O(dl)$. Traverse the bucket array and perform a total of dl key matching processes, resulting in a total time complexity of $\Theta(d^2l^2)$. The time complexity of the reordering and truncating process is the same as that of the top-k flow query, i.e. $\Theta(dl \cdot log(dl))$. Overall, the time complexity of querying the top-k applications is $\Theta(d^2l^2)$.

4.4. Top-k Heavy-sources

Top-k heavy sources mean the top-k sources who use the most traffic among online requesters in the network. Some of the heavy sources are people who like to watch movies, watch variety shows, and chase TV series. Other heavy sources are organizations that frequently download files on the Internet or frequently discuss online, which generates a lot of P2P traffic or audio traffic.

Algorithm 5. Query for Top-k Heavy-sources

Input: Querying number K
Output: Top-k heavy-sources $\mathcal{L}_{source}$
1: $\mathcal{L}_{source} = 0$
2: **for** $i = 1$ to d **do**
3: **for** $j = 1$ to l **do**
4: $\mathcal{L}_{source} \leftarrow (B(i,j).Full\_Key.SIP, B(i,j).Value)$
5: **end for**
6: **end for**
7: $ReorderAndTruncate(\mathcal{L}_{source})$
8: **return** $\mathcal{L}_{source}$

Algorithm 5 shows the query operation on top-k heavy-sources. The query for top-k heavy-sources and the query for top-k heavy-applications are both partial key queries, thus the query process is similar, differing only in that the extracted partial key is *SIP*. Similarly, the time complexity of the top-k heavy-sources query is $\Theta(d^2l^2)$.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of AppSketch compared with the alternative methods in terms of accuracy and efficiency. We first describe the real-world dataset used in our experiments. Then, we introduced the evaluation metrics and configurations of AppSketch in detail. We conducted all experiments on a Linux platform with an Intel Core i9-9900KF CPU (3.60GHz) and 32 GB DRAM memory. AppSketch and the alternative methods are implemented in C++ with publicly available at https://github.com/houchangsheng/AppSketch.

5.1. Datasets

The datasets we used were collected from a university campus network in Nanjing, Jiangsu Province of China. The datasets are pcap files with raw packets that were captured from a core switch of the campus network. The IP trace data were captured from 20:00 to 20:30 (i.e. one of the network traffic peaks of the day) on 26 November 2020. We divide the trace into five 6-min epochs, which contain 49.4 million packets on average. Note that we did not use open-source streaming datasets (e.g. CAIDA's IP-trace data [47]) as these datasets do not contain the payload of the packets such that the flow data can not be classified.

In order to accomplish real-time traffic analysis, the traffic classification method we employ relies on information such as port number, four-byte payload and flow statistics. It is important to note that due to the nature of encryption, our method may not be able to accurately identify all types of encrypted traffic. Encryption can effectively conceal the payload contents, making it challenging to classify such traffic accurately in real time. We acknowledge this limitation and are actively exploring ways to enhance our approach to handle encrypted traffic more effectively. In the campus network dataset, the number of unidentified encrypted TCP and UDP flows accounted for 0.7% and 3.8% of the total flows, respectively. Therefore the unrecognized encrypted traffic only takes up a tiny fraction (i.e. 4.5% in total) of the dataset. For the sake of statistics and representation, we remove this small portion of unidentified encrypted traffic from the dataset.

5.2. Metrics

5.2.1. *Jaccard Similarity*

Jaccard Similarity is used to compare the similarity and differences between two limited sample sets, where the value closer to one means a higher similarity. Given two flow sets A and B, their Jaccard Similarity is formalized as

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

5.2.2. *Normalized Discounted Cumulative Gain*

Normalized Discounted Cumulative Gain is used to evaluate the accuracy of the sorting results, referred to as NDCG for short. NDCG considers the factors of sort order, which makes the top-ranked items have higher gains and compromises the lower ranked items. Let the relevance score of the item ranked i in the list be $r(i)$, then Discounted Cumulative Gain of the list containing k items is formalized as

$$DCG = \sum_{i=1}^{k} \frac{r(i)}{\log_2(i+1)}.$$

NDCG is the normalized result of DCG.

5.2.3. *AAE and ARE*

Average Absolute Error (AAE) and Average Relative Error (ARE) are used to evaluate the accuracy of the estimate of the aggregate traffic on network flows and applications/sources traffic consumption. They are formalized as follows:

$$AAE(E_{results}) = \frac{\sum_{\forall x \in E_{results}} |\tilde{V}(x) - V(x)|}{|E_{results}|}$$

and

$$ARE(E_{results}) = \frac{\sum_{\forall x \in E_{results}} \frac{|\tilde{V}(x) - V(x)|}{V(x)}}{|E_{results}|}.$$

5.2.4. *Average Processing Time*

By comparing the average processing time, we evaluate the efficiency of AppSketch and other alternative methods. The average processing time refers to the average time it takes to process all incoming flows (insert all flows or return all top-k queries) in each epoch, which is specified as follows:

$$AverageProcessingTime = \frac{Total\ Processing\ Time}{Number\ of\ Epochs}.$$

5.3. Parameter Configurations

We implemented other alternative methods by changing the structure design and the updating strategy of the second stage of AppSketch. We compare with the latest sketch technologies used for heavy flow queries, including USS [42], DMatrix [44], WavingSketch [41], HeavyGuardian [40] and Cold Filter [38].

Multiple key queries are implemented in three ways:

- **Multiple-key sketch:** Multiple-key sketch supports queries for both full key and arbitrary partial keys. The multiple-key sketch is carefully designed for partial key queries, returning highly accurate partial key query results.
- **Aggregating the single-key sketch:** A single-key sketch is designed for querying a specific full key. When the single-key sketch returns the full key query results, the partial key query results can be obtained by aggregating the values of the relevant partial key. As single-key sketch is not dedicated to partial key queries, its aggregated results may have poor accuracy.
- **Using separate single-key sketches:** Consider a separate statistical method and construct several single-key sketches. Each sketch is only responsible for recording the statistical results corresponding to a certain kind of key. Supposing $\langle App, SIP \rangle$ is a full key, construct one single-key sketch for full key $\langle App, SIP \rangle$, and then construct one single-key sketch for partial keys $\langle App \rangle$ and $\langle SIP \rangle$, respectively. When querying, directly return the results from the corresponding sketch.

In our evaluation, we compared the accuracy of AppSketch with alternative methods, including USS, DMatrix, WavingSketch, HeavyGuardian and Cold Filter. Among these, AppSketch, USS and

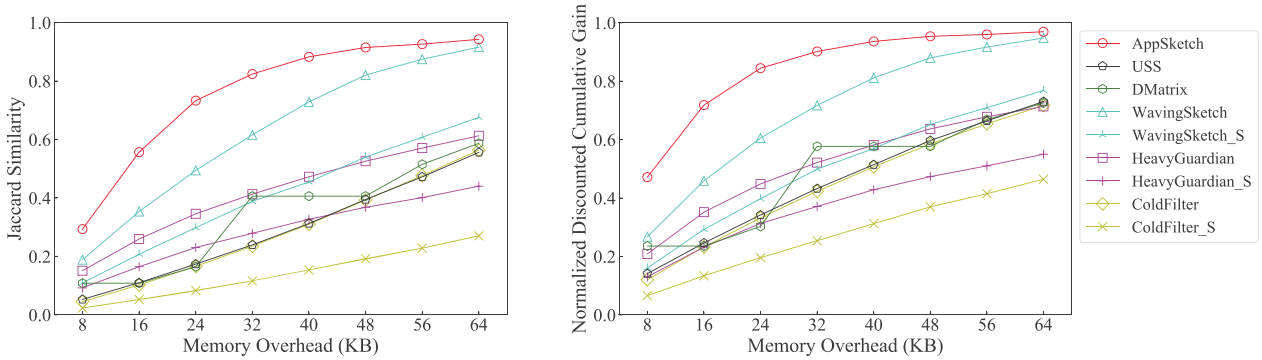

Figure 3. Jaccard Similarity (Left) and Normalized Discounted Cumulative Gain (Right) of the top-k heavy-flows query under varying memory overhead.

DMatrix are multiple-key sketches capable of directly returning partial key query results. On the other hand, WavingSketch, HeavyGuardian and Cold Filter are single-key sketches, and we compared the results of aggregating single-key sketch methods versus using separate single-key sketch methods. The separate single-key sketch methods are denoted as *WavingSketch_S*, *HeavyGuardian_S* and *ColdFilter_S*.

To evaluate accuracy, we considered varying memory overhead conditions. For sketches requiring multiple hash functions, we set the number of hash functions to 2 ($d = 2$). The USS implementation we evaluated is optimized with an enhanced update process utilizing a hash table and a double-linked list. DMatrix determines the bucket to record flow statistics by performing double hashing. To achieve more accurate partial key queries, we adjusted the length and width ratios of the DMatrix based on the ratio of the number of applications to the number of sources in the dataset.

For the three single-key sketches, except for the number of hash functions, we used the default parameter values provided in the respective papers. In the case of separate single-key sketches, we constructed individual sketches for heavy flows, heavy applications and heavy sources. The memory overhead was divided such that the heavy flow sketch occupied one-third of the total, and the remaining memory overhead was divided between the heavy application sketch and the heavy source sketch in the ratio of the number of applications to the number of sources. For the top-k queries, we set k to 1000 for top-k flows, 100 for top-k applications and 1000 for top-k sources.

5.4. Experimetal Results

5.4.1. *Top-k Heavy-flows Query*

We first evaluate the accuracy of top-k flows query for AppSketch and other methods. We compared the Jaccard Similarity and NDCG of top-k flows query and memory overhead changes from 8 to 64 KB. Figure 3 shows the Jaccard Similarity and the NDCG of top-k flows query. We can see that the Jaccard Similarity coefficients and NDCG of all methods' query results gradually increase as the memory overhead increases. This is because, for different methods, the relatively loose memory space makes hash conflicts less frequent or allows sketches to record more heavy flows. Clearly, AppSketch's top-k query performance outperforms the other methods. This indicates that AppSketch improves the performance of partial-key queries without losing the performance of full-key queries.

Next, we compare the average absolute error and the average relative error of top-k flows query. Figure 4 shows the average absolute error and the average relative error of top-k flows query. We can see that as the memory overhead increases, the average absolute error and the average relative error of all methods decrease gradually. This is reasonable, as for the same query task, the more loose memory space allows less bucket competition between flows and thus less effect between flows. The average absolute error of AppSketch is minimum for different memory overhead cases. However, the average relative error of AppSketch is not optimal. This is because AppSketch aims to minimize the estimation error of the subset sum, i.e. $\sum_{Key}(V(Key) - \hat{V}(Key))^2$. Such a design naturally makes AppSketch primarily minimize the absolute error rather than the relative error. Nevertheless, the average relative error of AppSketch query results is suboptimal when the memory overhead is greater than or equal to 24 KB.

5.4.2. *Top-k Heavy-applications Query and Top-k Heavy-sources Query*

We compared the Jaccard Similarity and NDCG of top-k applications query and top-k sources query. Memory overhead changes from 8 to 64 KB.

Figure 5 shows the Jaccard Similarity of top-k applications query and top-k sources query, respectively. For all methods, we can see that the trend of Jaccard Similarity coefficients for top-k applications query and top-k sources query is consistent with that of top-k flows query, gradually increasing with increasing memory overhead. In top-k applications query results, we find that AppSketch alternates with HeavyGuardian as the optimal performer as the memory overhead varies. However, in top-k sources query results, HeavyGuardian is far less accurate than AppSketch. When the memory overhead is greater than or equal to 48 KB, WavingSketch's top-k sources query is more accurate than AppSketch. This means that WavingSketch checks out more heavy sources. However, as shown in Fig. 6, AppSketch's performance is optimal when considering the the sort order of top-k sources.

Figure 6 shows the NDCG of top-k applications query and top-k sources query, respectively. We can see that the NDCG of all methods' query results gradually increases as the memory overhead increases, consistent with the top-k flows query. Meanwhile, the performance of AppSketch is always optimal as the memory overhead changes.

Next, we evaluate the estimation error of top-k applications query and top-k sources query. Figure 7 and Fig. 8 show the average absolute error and the average relative error of top-k applications query and top-k sources query. We can see that in most cases, AppSketch has the smallest average absolute error. Only when the memory overhead is 8 KB, AppSketch is slightly worse than *ColdFilter_S*, a separate single-key sketch method.

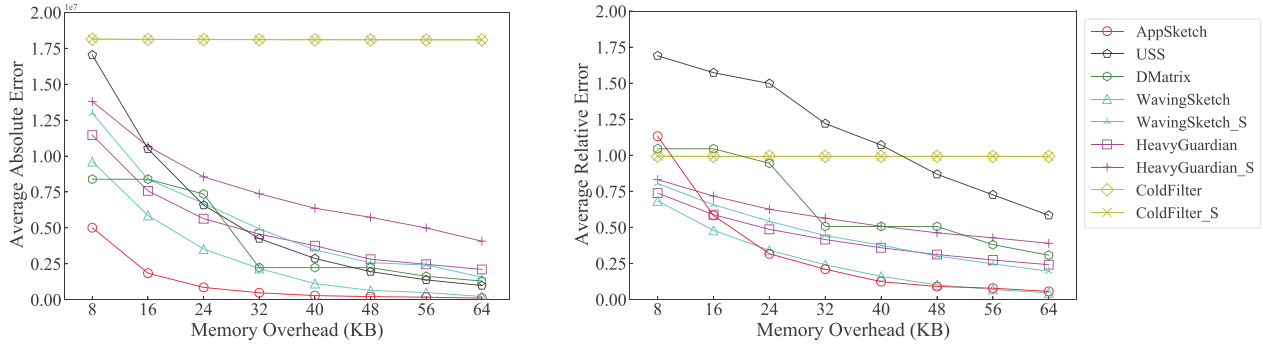

Figure 4. Average Absolute Error (Left) and Average Relative Error (Right) of the top-k heavy-flows query under varying memory overhead.

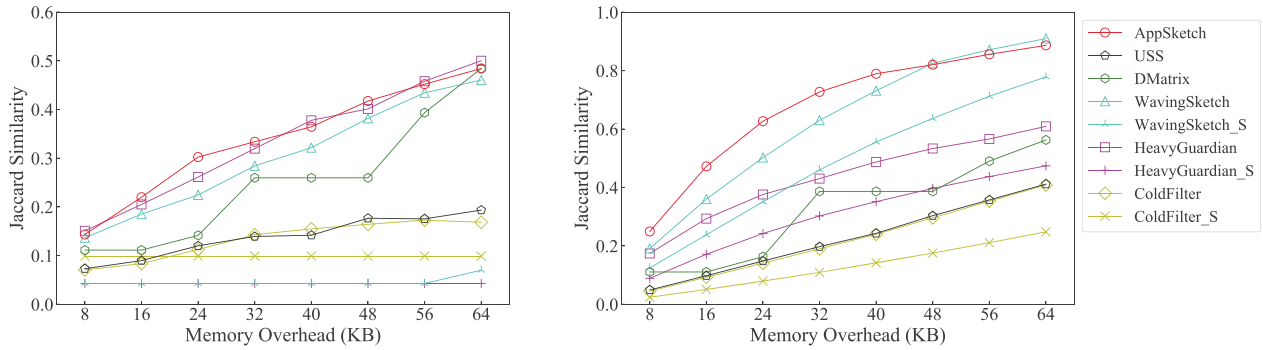

Figure 5. Jaccard Similarity of the top-k heavy-applications query (Left) and top-k heavy-sources query (Right) under varying memory overhead.

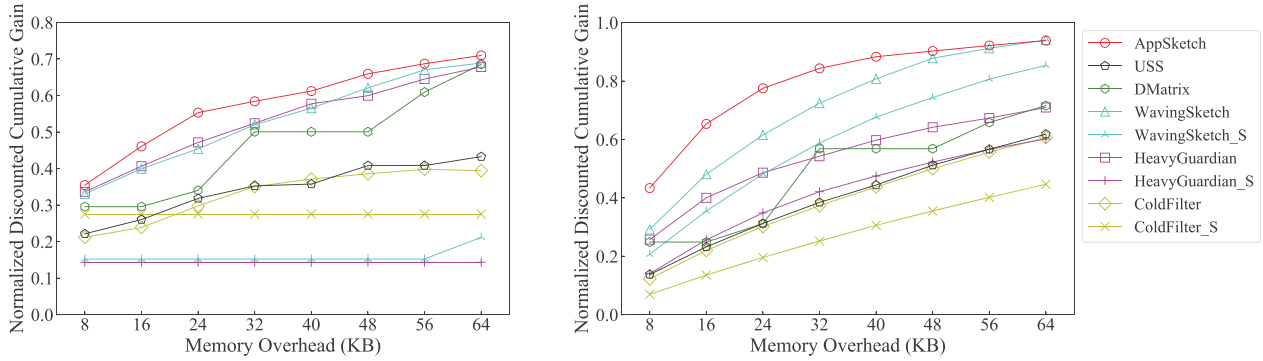

Figure 6. Normalized Discounted Cumulative Gain of the top-k heavy-applications query (Left) and top-k heavy-sources query (Right) under varying memory overhead.

This is because, when the memory space is compacted to 8 KB, AppSketch no longer has an advantage over the strategy of building a separate sketch for the partial key. Similar to the top-k flows query, the average relative error of AppSketch is not optimal. It is worth noting that for top-k applications query, the average relative error of the methods, except for the separate single-key sketch methods, shows a fluctuating trend as the memory overhead changes. This is due to the fact that there are only 123 applications in the dataset compared with a large number of sources (reaching more than 30 000). In this case, changing the recorded key probabilistically can easily affect the query results of a single partial key.

5.4.3. *Update and Query Efficiency*

Figure 9 shows the average time costs of the update and query processes in AppSketch and other alternative methods.

Note that as the memory overhead increases, there is a trend toward a gradual decrease in the update time of all methods. This is reasonable because the larger memory overhead results in fewer hash conflicts when updating in all methods, reducing the time required to execute additional processing mechanisms due to hash conflicts. In addition, as the memory overhead increases, there is a slight fluctuation (i.e. as the memory overhead increases, the update time also slightly increases) in the update time of methods such as AppSketch, DMatrix and ColdFilter, which is related to the basic principles of the relevant methods. For AppSketch, as memory overhead increases, the probability of two different keys mapping to the same bucket decreases, while it increases the possibility of the *Full Key* field of the smallest bucket fluctuating repeatedly. DMatrix is also in the dilemma of continuously updating candidate key fields. As the memory overhead increases, it becomes more difficult for ColdFilter to be filled with flows. However, when the two-layer data structure of ColdFilter is not fully filled, each update will perform more judgment and comparison operations.

As the memory overhead increases, the query time of all methods gradually increases, which is as expected. When the memory overhead increases, all methods can accommodate more heavy

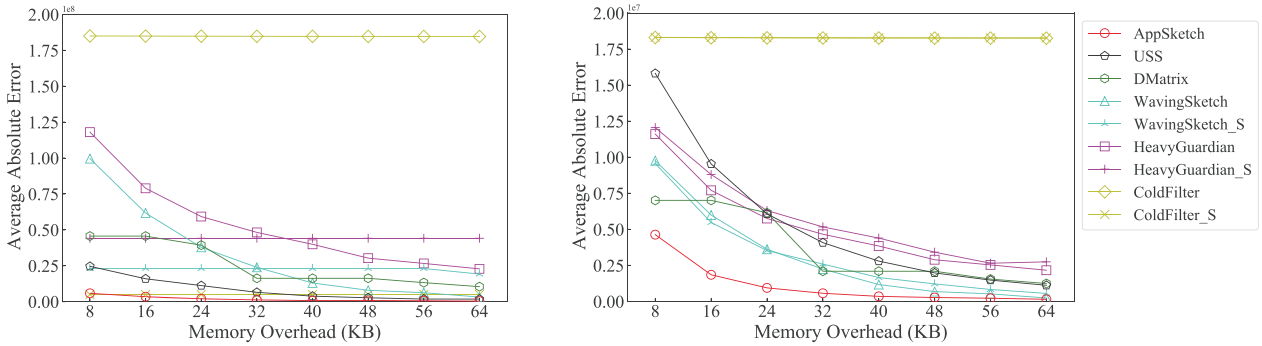

Figure 7. Average Absolute Error of the top-k heavy-applications query (Left) and top-k heavy-sources query (Right) under varying memory overhead.

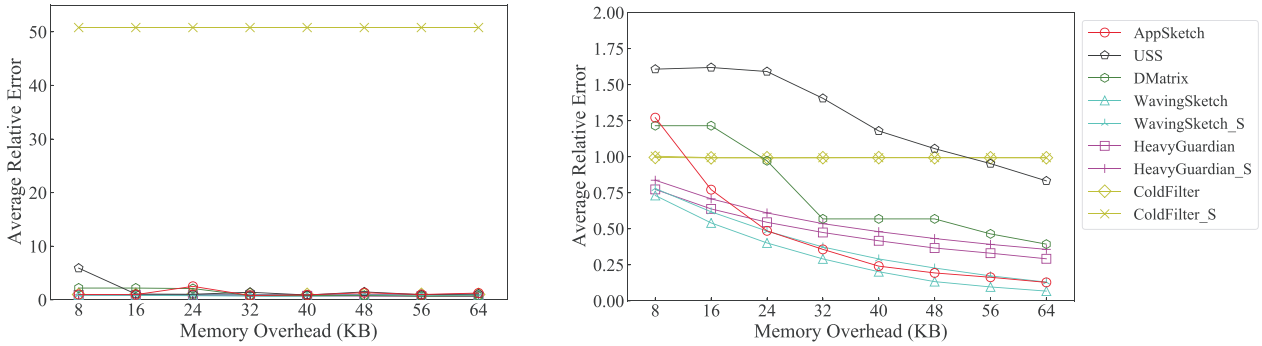

Figure 8. Average Relative Error of the top-k heavy-applications query (Left) and top-k heavy-sources query (Right) under varying memory overhead.

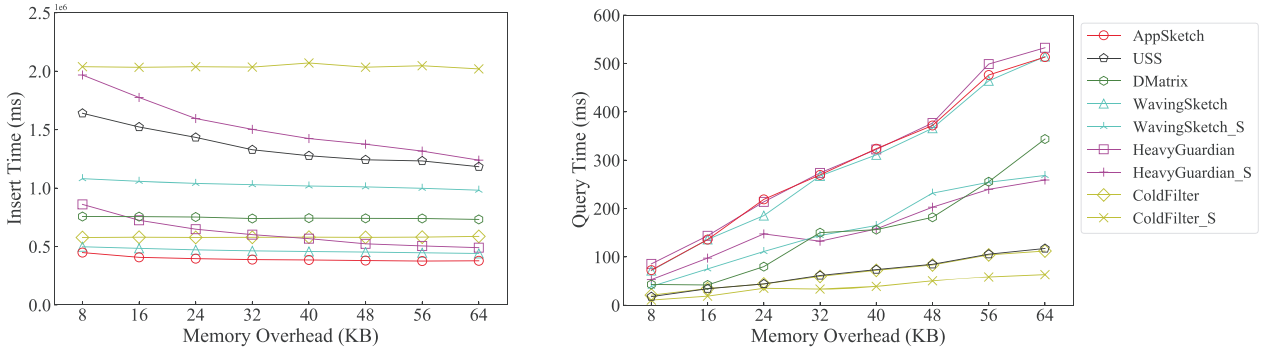

Figure 9. The average time cost of updating and querying process in AppSketch and other methods.

flows, resulting in more protracted times for querying all heavy flows (or finding top-k flows from heavy flows). In all cases, AppSketch has the least insert time, although its queries are not the fastest. However, we note that the insert time ranges from 3.8×10^5 to 2.1×10^6 ms, which is a very large order of magnitude compared with the query time that takes a maximum of 540 ms. Compared with other methods, AppSketch can store network flow information more quickly and conveniently, which is extremely important in online real-time processing, and the under-performance in query time is acceptable.

5.4.4. *Sensitivity*

We evaluate the sensitivity of AppSketch and other methods (DMatrix, ColdFilter) that require multiple hash functions to the number of hash functions in this section. We fixed the memory overhead to 64 KB and varied the number of hash functions d from 2 to 6.

Table 1 shows the Jaccard Similarity and the NDCG of several methods under varying the number of hash functions. We can see that, as d increases, the Jaccard Similarity and NDCG of all methods remain relatively stable, except for DMatrix, which exhibits severe fluctuations. This is because, for AppSketch and ColdFilter, more hash functions reduce the mutual influence between different flows, while suffering from other performance reductions due to unchanged memory overhead. For AppSketch, fixed memory overhead confines the total number of buckets $d \times l$, and as d increases, l correspondingly decreases, which increases the probability of different keys mapping to the same bucket (i.e. increases hash conflicts), resulting in a decrease in estimation performance. AppSketch, therefore, does not benefit significantly from increasing d. For DMatrix, due to its three-dimensional structure, an increase in d results in an increase in the number of layers, and the estimation performance on the two-dimensional table of each layer is significantly reduced. It is difficult to determine whether the advantage of reducing flow mutual influence can fully compensate for the impact of performance degradation in two-dimensional table estimation.

Table 1. Jaccard Similarity and Normalized Discounted Cumulative Gain of the top-k queries under varying the number of hash functions.

Query	Method	Jaccard Similarity					Normalized Discounted Cumulative Gain				
		$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
Top-k Heavy-flows	AppSketch	**0.948**	**0.976**	**0.985**	**0.989**	**0.993**	**0.971**	**0.987**	**0.992**	**0.993**	**0.995**
	DMatrix	0.573	0.523	0.584	0.432	0.281	0.715	0.687	0.739	0.594	0.444
	ColdFilter	0.563	0.592	0.606	0.606	0.606	0.717	0.728	0.727	0.727	0.727
	ColdFilter_S	0.273	0.279	0.282	0.282	0.282	0.467	0.472	0.470	0.470	0.470
Top-k Heavy-applications	AppSketch	**0.480**	**0.464**	**0.458**	**0.440**	**0.469**	**0.704**	**0.699**	**0.696**	**0.685**	**0.706**
	DMatrix	0.478	0.272	0.312	0.271	0.183	0.680	0.517	0.557	0.498	0.397
	ColdFilter	0.163	0.162	0.162	0.162	0.162	0.387	0.384	0.382	0.382	0.382
	ColdFilter_S	0.099	0.099	0.099	0.099	0.099	0.277	0.277	0.277	0.277	0.277
Top-k Heavy-sources	AppSketch	**0.883**	**0.925**	**0.939**	**0.950**	**0.960**	**0.936**	**0.961**	**0.969**	**0.974**	**0.978**
	DMatrix	0.552	0.471	0.531	0.398	0.260	0.707	0.651	0.703	0.575	0.441
	ColdFilter	0.408	0.425	0.434	0.434	0.434	0.606	0.617	0.619	0.619	0.619
	ColdFilter_S	0.248	0.254	0.256	0.256	0.256	0.446	0.451	0.451	0.451	0.451

Table 2. Average Absolute Error and Average Relative Error of the top-k queries under varying the number of hash functions.

Query	Method	Average Absolute Error($\times 10^7$)					Average Relative Error				
		$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
Top-k Heavy-flows	AppSketch	**0.013**	**0.003**	**0.002**	**0.002**	**0.001**	**0.058**	**0.024**	**0.015**	**0.012**	**0.009**
	DMatrix	0.130	0.125	0.097	0.311	0.526	0.320	0.416	0.368	0.647	0.987
	ColdFilter	1.770	1.772	1.774	1.774	1.774	0.992	0.993	0.994	0.994	0.994
	ColdFilter_S	1.771	1.774	1.775	1.775	1.775	0.993	0.994	0.995	0.995	0.995
Top-k Heavy-applications	AppSketch	**0.072**	**0.023**	**0.020**	**0.022**	**0.020**	0.763	1.037	1.225	**0.895**	**0.684**
	DMatrix	1.052	0.892	0.655	1.346	2.317	**0.699**	**0.870**	168.0	168.3	4.816
	ColdFilter	18.07	18.09	18.11	18.11	18.11	0.997	0.998	**0.999**	0.999	0.999
	ColdFilter_S	0.461	0.461	0.461	0.461	0.461	48.74	48.74	48.74	48.74	48.74
Top-k Heavy-sources	AppSketch	**0.018**	**0.006**	**0.004**	**0.004**	**0.003**	**0.128**	**0.074**	**0.057**	**0.049**	**0.043**
	DMatrix	0.126	0.124	0.096	0.258	0.458	0.397	0.497	0.453	0.684	1.247
	ColdFilter	1.787	1.789	1.791	1.791	1.791	0.993	0.994	0.995	0.995	0.995
	ColdFilter_S	1.789	1.791	1.793	1.793	1.793	0.993	0.995	0.996	0.996	0.996

In addition, there is a slight decrease in the performance of AppSketch and ColdFilter for top-k heavy-applications query. Since the dataset only contains 123 applications, one heavy application may correspond to a number of full key flows. After aggregation, the impact of increased hash conflicts is amplified and presented on the top-k heavy-applications query results. Overall, as d increases, the performance of AppSketch remains stable while outperforming all other methods.

Table 2 shows the average absolute error and average relative error of several methods under varying the number of hash functions. It can be seen that AppSketch and ColdFilter exhibit stability in estimation errors, while DMatrix still fluctuates significantly. For the same reason, increasing the number of hash functions also affects the performance of several methods in estimation error. In most cases, the average absolute error and average relative error of AppSketch are the minimum. For the top-k heavy-applications query, the average relative error of AppSketch is slightly worse than that of DMatrix and ColdFilter. The reason is that AppSketch aims to minimize the estimation error of the subset sum. Such a design naturally makes AppSketch primarily minimize the absolute error rather than the relative error.

Table 3 shows the average time cost of the updating and querying process of several methods under varying the number of hash functions. As d increases, the update time of several methods also increases, since this results in more hash operations. The query time of AppSketch and ColdFilter remains almost unchanged, while DMatrix shows a significant decrease. Increasing d does not affect the heavy-flows that AppSketch and ColdFilter can accommodate, while seriously compresses the heavy-flows that DMatrix can accommodate, and its query time also decreases as the number of queried flows decreases. The update efficiency of AppSketch is the highest, which is extremely important in online real-time processing, although the query efficiency is slightly poor.

Next, we evaluate the impact of the number of hash functions and memory overhead on AppSketch. Table 4 shows the Jaccard Similarity and the NDCG of AppSketch under varying the number of hash functions and memory overhead. It can be seen that increasing memory overhead can significantly improve the estimation accuracy performance of AppSketch compared with applying more hash functions. Furthermore, increasing d actually reduces the performance of top-k heavy-applications query of AppSketch.

Table 5 shows the average absolute error and average relative error of AppSketch under varying the number of hash functions and memory overhead. In top-k heavy-flows and top-k heavy-sources queries, as memory overhead increases or the

Table 3. The average time cost of the updating and querying process under varying the number of hash functions.

Method	Insert Time ($\times 10^5$ ms)					Query Time (ms)				
	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
AppSketch	**3.591**	**4.473**	**5.075**	**5.704**	**6.157**	526.36	508.44	510.30	506.37	511.53
DMatrix	6.942	9.454	11.825	14.338	16.726	323.31	191.71	237.67	193.77	127.79
ColdFilter	5.575	7.000	8.172	8.188	8.165	110.4	110.95	111.58	111.55	110.67
ColdFilter_S	19.663	23.408	26.886	26.821	26.946	**62.172**	**61.899**	**62.665**	**62.362**	**62.947**

Table 4. Jaccard Similarity and Normalized Discounted Cumulative Gain of the top-k queries in AppSketch under varying memory overhead and the number of hash functions.

Query	Memory Overhead	Jaccard Similarity					Normalized Discounted Cumulative Gain				
		$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
Top-k Heavy-flows	24 KB	0.730	0.802	0.825	0.846	0.850	0.844	0.890	0.904	0.917	0.920
	32 KB	0.826	0.880	0.902	0.924	0.935	0.904	0.936	0.949	0.960	0.965
	40 KB	0.871	0.925	0.947	0.963	0.968	0.929	0.960	0.972	0.980	0.983
	48 KB	0.906	0.954	0.970	0.978	0.984	0.949	0.976	0.984	0.988	0.991
	56 KB	0.922	0.965	0.979	0.986	0.986	0.958	0.981	0.988	0.992	0.992
	64 KB	0.948	0.976	0.985	0.989	0.993	0.971	0.987	0.992	0.993	0.995
Top-k Heavy-applications	24 KB	0.297	0.272	0.271	0.277	0.286	0.546	0.522	0.525	0.533	0.543
	32 KB	0.316	0.336	0.341	0.327	0.330	0.572	0.586	0.595	0.581	0.586
	40 KB	0.372	0.370	0.376	0.373	0.364	0.620	0.622	0.629	0.625	0.620
	48 KB	0.388	0.419	0.385	0.394	0.374	0.633	0.660	0.638	0.646	0.629
	56 KB	0.435	0.452	0.454	0.409	0.427	0.676	0.670	0.691	0.660	0.674
	64 KB	0.480	0.464	0.458	0.440	0.469	0.704	0.699	0.696	0.685	0.706
Top-k Heavy-sources	24 KB	0.623	0.673	0.698	0.707	0.714	0.773	0.812	0.829	0.835	0.839
	32 KB	0.713	0.766	0.788	0.806	0.803	0.837	0.871	0.884	0.895	0.896
	40 KB	0.779	0.832	0.841	0.865	0.871	0.877	0.910	0.916	0.929	0.933
	48 KB	0.829	0.871	0.899	0.903	0.912	0.906	0.932	0.947	0.950	0.955
	56 KB	0.847	0.907	0.923	0.941	0.941	0.917	0.951	0.959	0.969	0.969
	64 KB	0.883	0.925	0.939	0.950	0.960	0.936	0.961	0.969	0.974	0.978

number of hash functions increases, the estimation error of AppSketch gradually decreases. In the top-k heavy-applications query, the estimation error fluctuates within a small range, because one application may cover multiple different flows, and the heavy-applications query aggregates and amplifies the estimation error. Overall, expanding memory overhead or applying more hash functions does not result in poor performance of AppSketch in terms of estimation error, and in some cases, leads to better performance.

Table 6 shows the average time cost of the updating and querying process of AppSketch under varying the number of hash functions and memory overhead. As the memory overhead increases, the update time of AppSketch maintains a relatively stable state with a downward trend, while the query time gradually increases, which is a negligible order of magnitude compared with the update time. As d increases, the update time of AppSketch gradually increases, and the query time is relatively stable.

Overall, the high performance of AppSketch relies more on loose memory space constraints and is not sensitive to the number of hash functions. Specifically, applying more hash functions leads to increased update time, which is not conducive to online real-time flow processing.

5.4.5. *Memory Overhead*

We only analyze the memory overhead of AppSketch. When the memory overhead is 64KB, the NDCG of AppSketch reaches 0.97, 0.71 and 0.94 for top-k flows query, top-k applications query and top-k sources query, respectively. As the memory overhead gradually increases, the NDCG gradually approaches to 1; however, this will crowd out available memory for other systems or functions on the network device. In order to fit into the compact memory of the network device and to achieve real-time traffic analysis, it is suitable to request 64 KB of memory for AppSketch.

6. USE CASE

We analyzed the online behavior of college students using the stream data collected on the university campus network with AppSketch. The dataset was divided into five epochs, where each epoch contains a 6-min interval. For each epoch, we analyzed the top-k applications and top-k sources.

First, we analyzed the top-k application. Figure 10 shows the traffic size of the top-10 applications for five epochs and the traffic size change of applications that persist across five epochs. Note that HTTPS and HTTP applications took up most of the traffic in each epoch where HTTP and HTTPS are the primary protocol used for web browsing. However, HTTP/HTTPS protocol is also used by other applications which want to take advantage of the ubiquity features of the web browser. Such applications usually set a specific string for the User-Agent or Content-Type field so that the application software can identify their particular traffic. For example, Flash video streams will set the Content-Type

Table 5. Average Absolute Error and Average Relative Error of the top-k queries in AppSketch under varying memory overhead and the number of hash functions.

Query	Memory Overhead	Average Absolute Error ($\times 10^5$)					Average Relative Error				
		$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
Top-k Heavy-flows	24 KB	7.979	3.767	2.755	2.110	1.762	0.313	0.214	0.180	0.152	0.134
	32 KB	4.241	1.976	1.193	0.937	0.815	0.195	0.120	0.090	0.070	0.060
	40 KB	3.400	1.040	0.656	0.494	0.390	0.137	0.070	0.049	0.036	0.030
	48 KB	2.239	0.667	0.360	0.284	0.246	0.095	0.045	0.028	0.022	0.018
	56 KB	1.788	0.518	0.307	0.202	0.191	0.081	0.032	0.021	0.015	0.014
	64 KB	1.313	0.336	0.195	0.167	0.135	0.058	0.024	0.015	0.012	0.009
Top-k Heavy-applications	24 KB	19.77	8.654	8.029	5.759	5.067	1.167	0.871	0.861	0.943	0.779
	32 KB	9.846	5.843	4.124	3.369	3.381	0.722	1.196	1.094	1.510	2.573
	40 KB	10.39	3.996	3.030	2.461	2.376	1.040	1.784	0.740	1.611	0.782
	48 KB	9.075	3.607	2.294	2.242	1.868	0.978	2.983	0.762	0.642	0.699
	56 KB	6.828	3.068	2.124	1.958	2.319	0.687	0.912	2.325	0.639	0.710
	64 KB	7.150	2.350	2.036	2.184	1.952	0.763	1.037	1.225	0.895	0.684
Top-k Heavy-sources	24 KB	8.939	4.908	3.766	3.169	2.811	0.504	0.414	0.361	0.342	0.330
	32 KB	5.060	2.800	1.971	1.656	1.546	0.340	0.259	0.228	0.201	0.189
	40 KB	4.166	1.673	1.223	1.005	0.873	0.253	0.180	0.153	0.128	0.118
	48 KB	2.863	1.132	0.751	0.663	0.589	0.192	0.126	0.097	0.092	0.080
	56 KB	2.345	0.905	0.623	0.472	0.443	0.173	0.093	0.077	0.060	0.058
	64 KB	1.755	0.625	0.433	0.380	0.337	0.128	0.074	0.057	0.049	0.043

Table 6. The average time cost of the updating and querying process for AppSketch under varying memory overhead and the number of hash functions.

Memory Overhead	Insert Time ($\times 10^5$ ms)					Query Time (ms)				
	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
24 KB	4.000	4.933	5.783	6.508	7.169	216.762	215.451	209.924	212.094	211.689
32 KB	3.568	4.275	4.994	5.739	6.353	267.499	253.081	253.915	252.856	255.795
40 KB	3.726	4.531	5.247	5.978	6.549	337.203	312.91	314.536	318.01	314.641
48 KB	3.660	4.561	5.195	5.835	6.564	386.689	378.942	366.54	366.376	372.191
56 KB	3.677	4.470	5.021	5.738	6.254	492.217	477.423	469.982	468.842	464.202
64 KB	3.591	4.473	5.075	5.704	6.157	526.36	508.44	510.30	506.37	511.53

to 'video/flv'. Unfortunately, AppSketch cannot distinguish such applications from regular HTTP traffic because it only checks the first four bytes of the payload of packets. Therefore, HTTP/HTTPS traffic is an aggregation of many application traffic (e.g. video streams) in this case study. The QQLive is the second most traffic-consuming application, which is a video application of Tencent. In addition, we can see that YY_UDP, the traffic of YY live broadcast, is listed in the top-10 applications. This suggests that watching online videos or live broadcasts is very common among college students. There are also several applications involving P2P transmission. The STUN protocol stands for Simple Traversal of User Datagram Protocol (UDP) through Network Address Translators (NAT). This protocol is used in several different network implementations (e.g. VoIP). STUN is used to resolve the public IP of a device running behind a NAT, to solve problems such as one-way audio during a phone call or phone registration issues when trying to register to a VoIP or an IP PBX residing on a different network. Other P2P applications such as BitTorrent and Xunlei also consumed a lot of traffic within 30 min.

We analyzed the applications that continually appear in five epochs. Firstly, HTTPS and HTTP applications, which accounts for a relatively large proportion of all application traffic, continually appear in five epochs, and its traffic size shows significant differences among different epochs. There are two reasons for the fluctuation of HTTPS/HTTP traffic size: a large number of users widely use web browsers to access web pages, generating varying degrees of traffic at different time periods; HTTPS/HTTP protocol is used by other applications (such as video streaming applications), which generate traffic of different sizes due to inconsistent runtime. Secondly, QQLive and YY_UDP are typical audio and video traffic, characterized by roughly the same traffic size and persistence across different epochs. There are differences in the amount of traffic generated by different audio/video applications in each epoch. Furthermor, there are two P2P applications (STUN and BitTorrent_UDP) that maintain stable traffic sizes across multiple epochs, but have smaller traffic volume compared to video traffic.

Then, we analyzed top-k sources. Figure 11 shows the traffic size of the top five sources in each epoch. We observed a total of 17 different heavy sources in five epochs, where most of the top sources have a short duration of data transmission, while a small number of sources continuously transmit a large amount of traffic across multiple epochs, e.g. Source7 lasts 4 epochs. Table 7 shows the applications used by top five sources. HTTPS and HTTP traffic are still dominant. We obtained some speculative results of the traffic carried on HTTP by analyzing the destination of the

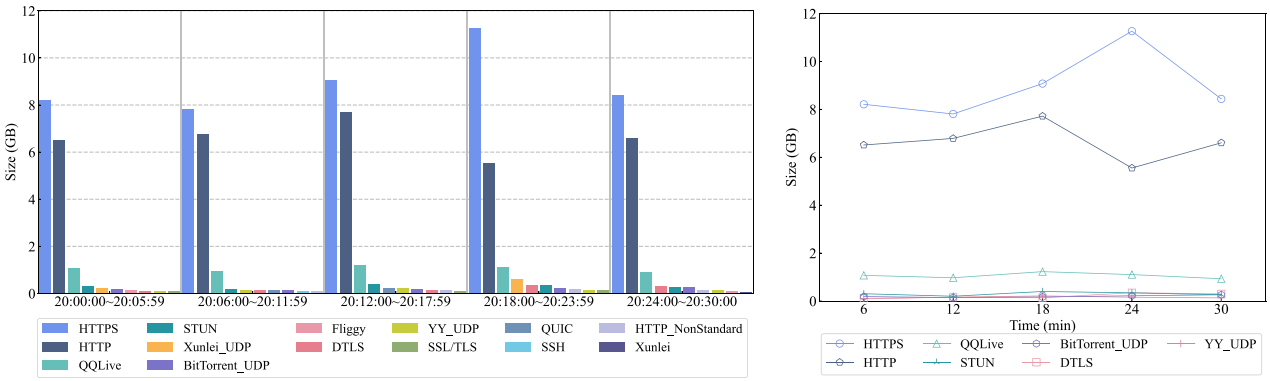

Figure 10. Top 10 applications traffic size of five epochs (Left) and the traffic size change of applications that persist across five epochs (Right).

Table 7. The proportion of applications used by top five sources

Applications	Size	Proportion
HTTPS	11.61GB	61.1513%
HTTP	7.30GB	38.4638%
QQLive	59.04MB	0.3109%
Fliggy	4.47MB	0.0236%
SSL/TLS	4.19MB	0.0220%
BaiduYunP2P	3.68MB	0.0194%
BitTorrent	1.05MB	0.0055%
QUIC	112.49KB	0.0006%
Taobao	97.78KB	0.0005%
ApplePush	87.32KB	0.0005%
Others	347.59KB	0.0018%

Table 8. The destination of heavy sources' HTTP traffic

Category	Destinations
Data Center	CERNET, Amazon, Qingdao Jiahua, Linode
Cloud Computing Center	Alibaba, Huawei, Google, NetEase, Baidu, Kingsoft
Content Delivery Network	EdgeCast, Wangsu, Akamai
University	Wuhan University of Technology, Hubei University of Technology
Internet Company	Baidu, Huawei, Microsoft
Network Operator	China Telecom, China Mobile, China Unicom
Other	—

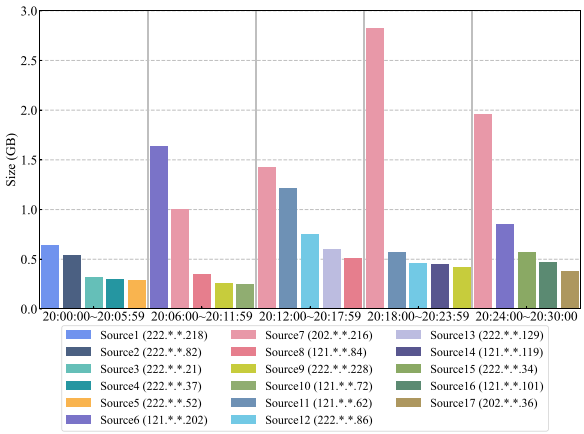

Figure 11. Top five sources traffic size of five epochs.

network flows, as shown in Table 8. First of all, we found that some heavy sources interact with data centers, including CERNET data centers, Amazon data centers, Qingdao Jiahua data centers, etc., which may be involved in the experimental data collection behavior of some research teams in the university. Secondly, some heavy sources communicate to the cloud resources of Internet companies (such as Alibaba, Huawei, Google and Kingsoft). College students also obtain resources from CDN nodes, including CND service providers such as EdgeCast, Wangsu and Akamai. Some heavy sources send data to destinations located in Wuhan University of Technology and Hubei University of Technology, which may be resource sharing among universities. Note that QQLive occupies 0.3% of the traffic consumed by heavy sources (as Table 7). Through analyzing top-k sources one by one, we found that Source8 (121.*.*.84) was using this application during this period.

The use case of campus network traffic analysis illustrates that AppSketch is effective and can be used to analyze traffic from different views (applications or sources), find heavy flows, heavy applications and heavy sources in the network. For different network management tasks, analysis results can help make decisions and optimize network performance.

7. CONCLUSION

Real-time traffic analysis is crucial for efficiently allocating network resources and providing optimal services. However, achieving real-time, fine-grained traffic analysis can be challenging. In this work, we introduce AppSketch, a novel sketch framework specifically designed for fast and accurate application traffic analysis. AppSketch offers a lightweight and real-time solution for analyzing network traffic and capturing detailed traffic statistics. Our framework employs a two-stage approach. In the first stage, network traffic is classified based on application types. Then, in the second stage, we utilize a multiple-key sketch to gather application-specific statistics. Through extensive experiments, we have validated that AppSketch outperforms existing solutions in terms of accuracy and efficiency. One advantage of AppSketch is its insensitivity to parameters such as the number of hash functions. Instead, it relies more on available memory space. With a memory overhead of 64 KB, AppSketch performs sufficiently and is suitable for network devices with limited memory capacity.

Furthermore, we have demonstrated the effectiveness of AppSketch through its application on real campus network

data streams. The results obtained from our experiments on the campus network data reveal interesting insights. We have identified HTTP/HTTPS and audio/video applications as the dominant heavy applications within the network. Furthermore, our analysis indicates that heavy sources primarily consist of users obtaining research data from the data center. These findings contribute to a deeper understanding of application traffic patterns and resource utilization within the network.

While AppSketch provides a significant step forward in application traffic analysis, there are several potential means for future research. One potential direction is to enhance AppSketch's capability to handle encrypted traffic, which poses a challenge due to the inability to inspect payload contents. Additionally, further investigation into optimizing the performance and resource utilization of AppSketch in large-scale network environments would be beneficial.

FUNDING

This work is supported by the National Natural Science Foundation of China [62072465, 62172155]; the China Postdoctoral Science Foundation [2023TQ0089]; and the Science and Technology Innovation Program of Hunan Province [2022RC3061, 2023RC3027].

DATA AVAILABILITY STATEMENT

The data underlying this article cannot be shared publicly for protecting user privacy in the real network traffic data. These real traffic data have the potential to reveal sensitive user information. The data will be shared on reasonable request. The source code of AppSketch and the underlying code used to perform the analyses described in this study can be found at https://github.com/houchangsheng/AppSketch.

REFERENCES

1. Cascarano, N., Ciminiera, L. and Risso, F. (2011) Optimizing deep packet inspection for high-speed traffic analysis. *J. Netw. Syst. Manage.*, **19**, 7–31.
2. Zhou, T., Cai, Z., Liu, F. and Su, J. (2023) In pursuit of beauty: aesthetic-aware and context-adaptive photo selection in crowdsensing. *IEEE Trans. Knowl. Data Eng.*, **35**, 9364–9377.
3. Liu, F., Zhang, M., Zheng, B., Cui, S., Ma, W. and Liu, Z. (2023) Feature fusion via multi-target learning for ancient artwork captioning. *Information Fusion*, **97**, 101811.
4. Guha, S. and McGregor, A. (2012) Graph synopses, sketches, and streams: a survey. *Proceedings of the VLDB Endowment*, **5**, 2030–2031.
5. Wang, H. and Li, B. (2020) Mitigating bottlenecks in wide area data analytics via machine learning. *IEEE Trans. Netw. Sci. Eng.*, **7**, 155–166.
6. Su, J., Zhao, B., Dai, Y., Cao, J., Wei, Z., Zhao, N., Song, C., Liu, Y. and Xia, Y. (2022) Technology trends in large-scale high-efficiency network computing. *Front. Inf. Technol. Electron. Eng.*, **23**, 1733–1746.
7. Fu, X., Liu, A., Xiong, N.N., Wang, T. and Zhang, S. (2023) ATWR-SMR: an area-constrained truthful worker recruitment based sensing map recovery scheme for sparse MCS in extreme-environment internet-of-things. *IEEE Internet Things J.*, **10**, 1–14.
8. Chen, L., Gao, S., Liu, B., Lu, Z. and Jiang, Z. (2020) FEW-NNN: a fuzzy entropy weighted natural nearest neighbor method for flow-based network traffic attack detection. *China Commun.*, **17**, 151–167.
9. Xie, Z., Li, Z., Gui, J., Liu, A., Xiong, N.N. and Zhang, S. (2023) UWPEE: using UAV and wavelet packet energy entropy to predict traffic-based attacks under limited communication, computing and caching for 6G wireless systems. *Future Gener. Comput. Syst.*, **140**, 238–252.
10. Sharon, Y., Berend, D., Liu, Y., Shabtai, A. and Elovici, Y. (2022) TANTRA: timing-based adversarial network traffic reshaping attack. *IEEE Trans. Inf. Forensics Secur.*, **17**, 3225–3237.
11. Zhu, Y., Cui, L., Ding, Z., Li, L., Liu, Y. and Hao, Z. (2022) Black box attack and network intrusion detection using machine learning for malicious traffic. *Comput. Secur.*, **123**, 102922.
12. Lin, Z., Shi, Y., Xue, Z. (2022) IDSGAN: generative adversarial networks for attack generation against intrusion detection. In Gama, J., Li, T., Yu, Y., Chen, E., Zheng, Y., Teng, F. (eds), *Advances in Knowledge Discovery and Data Mining*. Springer, Cham.
13. Antonello, R., Fernandes, S., Kamienski, C., Sadok, D., Kelner, J., Gódor, I., Szabó, G. and Westholm, T. (2012) Deep packet inspection tools and techniques in commodity platforms: challenges and trends. *J. Netw. Comput. Appl.*, **35**, 1863–1878.
14. Shahraki, A., Abbasi, M., Taherkordi, A. and Jurcut, A.D. (2022) A comparative study on online machine learning techniques for network traffic streams analysis. *Comput. Netw.*, **207**, 108836.
15. Nguyen, T.T.T., Armitage, G., Branch, P. and Zander, S. (2012) Timely and continuous machine-learning-based classification for interactive ip traffic. *IEEE/ACM Trans. Netw.*, **20**, 1880–1894.
16. Wang, Y., Papageorgiou, M. and Messmer, A. (2006) A real-time freeway network traffic surveillance tool. *IEEE Trans. Control Syst. Technol.*, **14**, 18–32.
17. Yang, C.-T., Chen, S.-T., Liu, J.-C., Yang, Y.-Y., Mitra, K. and Ranjan, R. (2019) Implementation of a real-time network traffic monitoring service with network functions virtualization. *Future Gener. Comput. Syst.*, **93**, 687–701.
18. Rouhani, B. D., Songhori, E. M., Mirhoseini, A., and Koushanfar, F. (2015) SSketch: an automated framework for streaming sketch-based analysis of big data on FPGA. *Proceedings of IEEE annual international symposium on field-programmable custom computing machines (FCCM)*, Vancouver, BC, 2–6 may, pp. 187–194. IEEE, New York, NY.
19. Wang, W., Sun, Y., Zheng, K., Kaafar, M. A., Li, D., and Li, Z. (2014) Freeway: adaptively isolating the elephant and mice flows on different transmission paths. *Proceedings of IEEE international conference on network protocols*, Raleigh, NC, 21–24 October, pp. 362–367. IEEE, New York, NY.
20. Han, H., Yan, Z., Jing, X. and Pedrycz, W. (2022) Applications of sketches in network traffic measurement: a survey. *Information Fusion*, **82**, 58–85.
21. Wellem, T., Lai, Y.-K., Huang, C.-Y. and Chung, W.-Y. (2019) A flexible sketch-based network traffic monitoring infrastructure. *IEEE Access*, **7**, 92476–92498.
22. Cormode, G. and Muthukrishnan, S. (2005) An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, **55**, 58–75.
23. Fathi-Kazerooni, S. and Rojas-Cessa, R. (2021) Countering machine-learning classification of applications by equalizing network traffic statistics. *IEEE Trans. Netw. Sci. Eng.*, **8**, 3392–3403.
24. Alcock, S. and Nelson, R. (2012) *Libprotoident: Traffic classification using lightweight packet inspection categories and subject descriptors*. WAND Network Research Group, Hamilton, New Zealand.
25. OpenDPI. [Online]. http://www.opendpi.org.
26. L7-filter. [Online]. http://l7-filter.clearfoundation.com.

27. Alcock, S. and Nelson, R. (2013) Measuring the accuracy of open-source payload-based traffic classifiers using popular internet applications. *Proceedings of the IEEE international conference on local computer networks workshops (LCN workshops)*, Sydney, NSW, 21-24 October, pp. 956–963. IEEE, New York, NY.
28. Carela-Español, V., Bujlow, T., Barlet-Ros, P. (2014) Is our ground-truth for traffic classification reliable? In: Faloutsos, M., Kuzmanovic, A. (eds), *Passive and Active Measurement*. Springer, Cham.
29. Alcock, S., Möller, J.-P., and Nelson, R. (2016) Sneaking past the firewall: quantifying the unexpected traffic on major tcp and udp ports. *Proceedings of the ACM internet measurement conference (IMC)*, Santa Monica, CA, 14-16 November, pp. 231–237. ACM, New York, NY.
30. Charikar, M., Chen, K. and Farach-Colton, M. (2004) Finding frequent items in data streams. *Theor. Comput. Sci.*, **312**, 3–15.
31. Krishnamurthy, B., Sen, S., Zhang, Y., and Chen, Y. (2003) Sketch-based change detection: methods, evaluation, and applications. *Proceedings of the ACM internet measurement conference (IMC)*, Miami Beach, FL, 27-29 October, pp. 234–247. ACM, New York, NY.
32. Huang, Q., Jin, X., Lee, P. P. C., Li, R., Tang, L., Chen, Y.-C., and Zhang, G. (2017) SketchVisor: robust network measurement for software packet processing. *Proceedings of the conference of the ACM special interest group on data communication (SIGCOMM)*, Los Angeles, CA, 21-25 august, pp. 113–126. ACM, New York, NY.
33. Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., and Uhlig, S. (2018) Elastic sketch: adaptive and fast network-wide measurements. *Proceedings of the conference of the ACM special interest group on data communication (SIGCOMM)*, Budapest, Hungary, 20-25 august, pp. 561–575. ACM, New York, NY.
34. Liu, Z., Ben-Basat, R., Einziger, G., Kassner, Y., Braverman, V., Friedman, R., and Sekar, V. (2019) Nitrosketch: robust and general sketch-based monitoring in software switches. *Proceedings of the ACM special interest group on data communication (SIGCOMM)*, Beijing, 19-23 august, pp. 334–350. ACM, New York, NY.
35. Sheng, S., Huang, Q., Wang, S. and Bao, Y. (2021) PR-sketch: monitoring per-key aggregation of streaming data with nearly full accuracy. *Proceedings of the VLDB Endowment*, **14**, 1783–1796.
36. Huang, Q., Sheng, S., Chen, X., Bao, Y., Zhang, R., Xu, Y., and Zhang, G. (2021) Toward nearly-zero-error sketching via compressive sensing. *Proceedings of the USENIX symposium on networked systems design and implementation (NSDI)*, Boston, MA, 12-14 April, pp. 1027–1044. USENIX association, Berkeley, CA.
37. Zhao, Y. et al. (2021) LightGuardian: a full-visibility, lightweight, in-band telemetry system using sketchlets. *Proceedings of the USENIX symposium on networked systems design and implementation (NSDI)*, Boston, MA, 12-14 April, pp. 991–1010. USENIX association, Berkeley, CA.
38. Zhou, Y., Yang, T., Jiang, J., Cui, B., Yu, M., Li, X. and Uhlig, S. (2018) Cold filter: a meta-framework for faster and more accurate stream processing. *Proceedings of the international conference on Management of Data (SIGMOD)*, Houston, TX, 10-15 June, pp. 741–756. ACM, New York, NY.
39. Metwally, A., Agrawal, D., El Abbadi, A. (2004) Efficient computation of frequent and top-k elements in data streams. In: Eiter, T., Libkin, L. (eds), *Database Theory – ICDT 2005*. Springer, Berlin, Heidelberg.
40. Yang, T., Gong, J., Zhang, H., Zou, L., Shi, L. and Li, X. (2018) HeavyGuardian: separate and guard hot items in data streams. *Proceedings of the international conference on knowledge discovery and data mining (KDD)*, London, 19–23 august, pp. 2584–2593. ACM, New York, NY.
41. Li, J., Li, Z., Xu, Y., Jiang, S., Yang, T., Cui, B., Dai, Y. and Zhang, G. (2020) Waving sketch: an unbiased and generic sketch for finding top-k items in data streams. *Proceedings of the international conference on knowledge discovery and data mining (KDD), virtual event, CA*, 6-10 July, pp. 1574–1584. ACM, New York, NY.
42. Ting, D. (2018) Data sketches for disaggregated subset sum and frequent item estimation. *Proceedings of the international conference on Management of Data (SIGMOD)*, Houston, TX, 10-15 June, pp. 1129–1140. ACM, New York, NY.
43. Zhang, Y., Liu, Z., Wang, R., Yang, T., Li, J., Miao, R., Liu, P., Zhang, R. and Jiang, J. (2021) Coco sketch: high-performance sketch-based measurement over arbitrary partial key query. *Proceedings of the international conference on applications, technologies, architectures, and protocols for computer communication (SIGCOMM), virtual event, CA*, 23-27 august, pp. 207–222. ACM, New York, NY.
44. Hou, C., Hou, B., Zhou, T. and Cai, Z. (2021) DMatrix: toward fast and accurate queries in graph stream. *Comput. Netw.*, **198**, 108403.
45. Jing, X., Yan, Z., Jiang, X. and Pedrycz, W. (2019) Network traffic fusion and analysis against ddos flooding attacks with a novel reversible sketch. *Information Fusion*, **51**, 100–113.
46. Jing, X., Yan, Z., Han, H. and Pedrycz, W. (2022) Extended sketch: fusing network traffic for super host identification with a memory efficient sketch. *IEEE Trans. Dependable Secure Comput.*, **19**, 3913–3924.
47. CAIDA's IP-trace datasets, San Diego Supercomputer Center, San Diego. [Online]. https://www.caida.org/catalog/datasets/passive_dataset/.