Dissertation

Towards Scalable Network Traffic Measurement with Sketches

by

Rhongho Jang

Date: June 4, 2020

Department of Computer Science
University of Central Florida
Orlando, FL 32816

Doctoral Committee:
Dr. Aziz Mohaisen, Chair
Dr. Damla Turgut
Dr. Murat Yoksel
Dr. Sung Choi Yoo
Dr. Wei Zhang

Abstract

We are inching closer to the zettabyte era with the ever-increasing volumes of traffic on the Internet. By 2023, there will be around 100 zettabytes of data (Source: EMC). The increasing data volumes not only accelerated the development of processing, storage, and I/O devices but also the network infrastructure. As of now, the per-port speed of network devices reached 100 Gbps, and high-end switches are capable of processing more than 25.6 Tbps of network traffic. As one of the key functionality of such devices, network traffic measurement is crucial in many fields, such as billing, load balancing, anomaly detection, intrusion detection, and network failure detection. However, network traffic measurement is still at an early stage and facing various challenges. In practice, traffic measurement relies on either sampling or advanced devices. To ensure online processing, Cisco's NetFlow maintains flow information in TCAM and their statistics in SRAM. However, the number of entries in the table cannot be large because those memory chips are expensive. Instead, sFlow sends the collected packet headers (i.e., samples) periodically to a collecting server over the network to minimize the overhead in the data-plane. However, it presents a Control Loop between the server and switch, which leads to inaccurate analysis and delayed detection/response. To this end, any measurement system falls into either one of these two models.

For fast and accurate traffic measurement, managing an accurate working set of active flows (WSAF) from massive volumes of packet influxes at line rates is a key challenge. WSAF is usually located in high-speed but expensive memory, such as TCAM or SRAM, and thus the number of entries to be stored is quite limited. To cope with the scalability issue of WSAF, in the first phase of this dissertation, we propose to use In-DRAM WSAF with scales, and put a compact data structure called FlowRegulator in front of WSAF to compensate for DRAM's slow access time by substantially reducing massive influxes to WSAF without compromising measurement accuracy. To verify its practicability, we further build a per-flow measurement system, called InstaMeasure, on an off-the-shelf Atom (lightweight) processor board. We evaluate our proposed system by a large scale real-world experiment (connected to monitoring port of our campus main gateway router for 113 hours, and capturing 122.3 million flows). We verify that InstaMeasure can detect heavy hitters (HH) with 99% accuracy and within 10 ms (detection is faster for heavier HHs) while providing the one million flows record with only tens of megabytes of DRAM memory. InstaMeasure's various performance metrics are further investigated by the packet trace-driven experiment using one-hour CAIDA dataset, where the target of measurement was all the 78 million L4 flows for one-hour.

For the second part of this dissertation, and system-wide, we propose an SDN-based WLAN monitoring and management framework called RFlow^{+} to address WiFi service dissatisfaction caused by the limited view (lack of scalability) of network traffic monitoring and

absence of intelligent and timely network treatments. Existing solutions (e.g., OpenFlow and sFlow) have a limited view, no generic flow description, and a poor trade-off between measurement accuracy and network overhead depending on the selection of the sampling rate. To resolve these issues, we devise a two-level counting mechanism, namely a distributed local counter (on-site and real-time) and central collector (a summation of local counters). With this, we proposed a highly scalable monitoring and management framework to handle immediate actions based on short-term (*e.g.* 50 ms) monitoring and eventual actions based on long-term (*e.g.* 1 month) monitoring. The former uses the local view of each access point (AP), and the latter uses the global view of the collector. Experimental results verify that RFlow$^+$ can achieve high accuracy (less than 5% standard error for short-term and less than 1% for long-term) and fast detection of flows of interest (within 23 ms) with manageable network overhead. We prove the practicality of RFlow$^+$ by showing the effectiveness of a MAC flooding attacker/a super-spreader quarantine in a real-world testbed.

In the third and last piece of this dissertation, we aim to design a novel sampling scheme to deal with the poor trade-off provided by random sampling. Sampling is a powerful tool to reduce the processing overhead in various systems. NetFlow uses a local table for counting records per flow, and sFlow sends out the collected packet headers periodically to a collecting server over the network. Any measurement system falls into either one of these two models. To reduce the overhead, as in sFlow, simple random sampling (SRS) has been widely used in practice because of its simplicity. However, SRS provides non-uniform sampling rates for different fine-grained flows (defined by 5-tuple), because it samples packets over an aggregated data flow (defined by switch port or VLAN). Consequently, some flows are sampled more than the designated sampling rate (resulting in over-estimation), and others are sampled fewer (resulting in under-estimation). Starting with a simple idea that "independent per-flow packet sampling provides the most accurate estimation of each flow", we introduce a new concept of per-flow systematic sampling, aiming to provide the same sampling rate across all flows. In addition, we provide a concrete sampling method called SketchFlow, which approximates the idea of the per-flow systematic sampling using a sketch saturation event. We demonstrate SketchFlow's performance in terms of accuracy, sampling rate, and overhead using real-world datasets, including a backbone network trace, I/O trace, and Twitter dataset. Experimental results show that SketchFlow outperforms SRS (*i.e.* sFlow) and the non-linear sampling method while requiring a small CPU overhead to measure high-speed traffic in real-time.

Contents

List of Figures

List of Tables

1 INTRODUCTION

We are inching closer to the zettabyte era with ever-increasing volumes of traffic on the Internet. According to a Cisco's report [13], the annual Internet traffic will reach 3.3ZB per year by 2021. To deal with the rapidly surging demands on network bandwidth, per-port bandwidth now reaches 100 Gbps, or even more. To improve the utilization of the deployed network equipment (e.g., switch and router) by traffic engineering and secure networks, the role of traffic measurement becomes more important than ever, especially for data centers, where large volumes of traffic are moved between different sites or even with a single datacenter. Therefore, to enable fine-grained network traffic control, per-flow measurement (5-tuple: source IP address/port number, destination IP address/port number, and protocol) and its treatment become more crucial. Thanks to the high-speed network traffic, measurement algorithms now have to cope with enormous incoming data rates (*i.e.* larger number of flows) with tight deadlines (*i.e.* real-time). We stress that instant measurement is highly necessary for the data center traffic engineering (TE) and network anomaly detection. For example, if denial of service (DoS) attack causes an influx of packets at 100 Gbps, the detection delay of 100 ms will cause 1.2GB data to hit a server or a network. Therefore, to eschew large bandwidth payment, instant anomaly detection is essential. The similar issue happens also in the wireless domain. With the plethora of WLAN deployments in residential and enterprise settings, the Internet has become more accessible than ever. This proliferation has become even more expedited because of increasing demands from a wide range of user devices, e.g., smartphones and tablet PCs. In order not to lag behind users' aggressive network bandwidth demands in their daily lives (e.g., for YouTube or Netflix), WLAN technologies have rapidly advanced: 802.11n (up to 600 Mbps), 802.11ac (up to 6.933 Gbps), and so on [89].

1.1 Statement of Research

Limitation of Existing Sketch Algorithms. For per-flow measurement, sketch-based techniques have been greatly enhanced over several decades, starting with original proposals such as Flajolet-Martin (FM) sketch and Alon *et al.*'s approximate frequency measurement [1, 25] [15, 20, 24, 50, 54, 58, 78]. Unlike their counterparts (e.g., Netflow [12], sflow [85], jflow [43], etc.), sketch-based counting algorithms only require a small amount of memory to measure a large volume of traffic in real-time. To decrease memory usage, most works have used statistically shared counters [54], matrices [25], and Bloom filters [78] as statistical noise from each estimation can be removed at the time of estimation (or decoding). To enhance estimation accuracy, maximum likelihood estimation is usually adopted, thereby introducing a substantial amount of additional computations.

Due to their designs, most of the sketch-based decoding algorithms involve hundreds of hash calculations (*i.e.* computationally hard) and memory accesses from statistically mixed random blocks [36] to obtain meaningful statistics (e.g., heavy hitters, DDoS attack, flow size distribution and entropy, etc.) [54, 58, 78]. For this reason, offline decoding in a high-performance server is commonly accepted in practice but inherently incurs huge network delay. Particularly, for a software switch that is to be wildly used in a data center server, remote decoding undoubtedly increases the network congestion which degrades the user experience. Thus, online decoding is highly necessary for instant measurement and further timely detection. To enable instant measurements, scalability, as well as online decoding of measurement algorithms, are essential. This is because sketches are quickly saturated, and cannot count anymore when a flow grows, forcing the saturated sketch to be sent to a remote collector over the network and resulting in a high detection latency. For the scalability, instead of sending out a saturated sketch to a collector, we can decode and store the value into a table in a switch (or router) for hours or even days. By doing that, a switch can always refer to the table that keeps track of flows and their sizes. However, this approach requires not only online decoding capability of the underlying sketch, but also the scalability of the table, because our target time scale is very long—an hour to a week. Naturally, we considered the working set of active flows (hereafter, WSAF), which should be maintained by a switching fabric/software (*e.g.* Openswitch) for measurement and further refinements (*e.g.* routing, TE, and so forth). A WSAF is a type of cache of a full flow table, which can be found usually in TCAM (Ternary Content Addressable Memory), CAM, or sometimes SRAM for fast switching (or forwarding) [18]. NetFlow uses TCAM for storing WSAF in which an entry consists of a flow ID and the counting value, while OpenSketch takes advantage of TCAM and SRAM [12,98]. The number of entries in the table cannot be large because those types of memories are quite expensive.

Limitation of Existing Monitoring Systems. System-wise, interestingly but unfortunately, despite the advancements of WLAN technologies, people are easily dissatisfied with their WLAN infrastructures. The reasons for this dissatisfaction are two-fold: (1) an absence of intelligent and timely network treatments followed by (2) the limited view of network traffic monitoring tools (e.g., NetFlow [12] and sFlow [85]) and vendor-oriented configurability. Instead of näive over-provisioning of access points (APs), we can provide users with more stable and thus more reliable network conditions (e.g., latency, jitter, and required minimum bandwidth) by accurate network monitoring and timely treatments such as rate-limiting, the access control list (ACL), or flow quarantines.

Recently, intense efforts in two main streams have been made to realize the "*victory*" of SDN-driven data centers like B4 [39] in the WAN domain. First, efforts have been made in WLAN management frameworks. Unlike OpenFlow [60], a *de facto* standard interface

between a controller and switches, a WLAN management framework requires additional features such as wireless channel selection, interference mitigation, and mobility management. To achieve these, BeHop [95], Odin [82], and OpenSDWN [81] customized OpenFlow's configurability for WLAN by introducing the concept of virtual APs. The other optimization efforts have addressed WLAN monitoring frameworks (e.g., PayLess [11], OpenSketch [98], FlowSense [97], and OpenTM [90]). These monitoring frameworks tried to overcome the intrinsic limitations (*i.e.* the limited accuracy of default settings and resource-hungry nature of full sampling) of generic sampling based solutions—NetFlow [12] and sFlow [85].

Unlike backbone network, WLAN requires a network management framework to monitor wireless network traffic at different target levels (*i.e. short-term* bursty users and *long-term* heavy down-loaders/up-loaders) because of its openness. Also, the management framework needs to improve overall wireless bandwidth utilization by timely resource allocation actions (*i.e. immediate* action according to short-term monitoring results and *eventual* action according to long-term monitoring) as well as accommodate more users by dynamically providing capacity. To our best knowledge, no existing studies have ever included both approaches in its design considerations.

Limitation of Sampling Algorithm. The bottleneck of NetFlow is the processing capacity for the local table, and that of sFlow is the network capacity. To address the bottleneck, the widely-adopted simple random sampling (SRS) is used with a very small overhead. In theory, SRS guarantees each packet has an equal chance to be sampled. However, the general usage of SRS is for sampling over the interface or VLAN, which collects coarse samples without considering the individual fine-grained flows, such as a flow defined by the 5-tuple. Consequently, some flows are sampled more than the designated sampling rate, resulting in over-estimation, while others suffer from under-estimation. We note that, although the main purpose of traffic measurement is mostly to obtain per-flow statistics such as the spectral density of flow size and distribution, sampling has been applied to data streams aggregating all the flows, rather than individual flows. SRS samples packets with $1/p$ over the entire data stream, although it cannot guarantee the sampling rate to be $1/p$ for each flow.

1.2 Approaches

InstaMeasure. Algorithm-wise, to support scalability by increasing the WSAF's capacity, we can put WSAF in DRAM instead of the expensive memory (*i.e.* incentive to cost-effectiveness). However, there is a speed issue for In-DRAM WSAF: a packet arrival rate is too fast to handle by In-DRAM WSAF, owing to the DRAM's speed and WSAF table's hash collision. Unfortunately, most sketch-based algorithms lack scalability and online decoding capabilities. Our approach to solving these two problems is 1) to use a counting algorithm

that can perform online decoding and 2) to put a flow regulator before WSAF to slow down the incoming packet rate to WSAF. To realize both ideas, we designed a highly scalable counting and flow regulating algorithm called FlowRegulator. By design, instead of directly inserting or updating every packet of flow into the WSAF table, FlowRegulator (*i.e.* a small cache buffer) retains a fraction of flow counts. By doing so, we can suppress frequent WSAF updates in DRAM; thereby FlowRegulator can support the large-scale influx of flows with the use of cost-effective large DRAM. Consequently, FlowRegulator relaxes the necessity of precious memories (TCAM or SRAM) for maintaining large WSAF, and further enables us to build a highly scalable and fast measurement system. We realize that in a system called InstaMeasure.

RFlow$^+$. System-wise, we propose RFlow$^+$ to achieve two different levels of network monitoring—local (switch/AP level) and global (controller/collector level); thereby supporting application-specific actions (*i.e.* *immediate* and *eventual*) via a network management framework. The recyclable counter with confinement (RCC) [66] motivates RFlow$^+$'s major design as it provides reliable counting accuracy while efficiently managing its memory usage; this consequently reduces network overheads, as further detailed in Section **??**. Mainly because of our two-level (*i.e.* global and local) monitoring framework design based on the RCC counter, RFlow$^+$'s major departure from existing work is that the *local agent* takes the first step toward supporting immediate actions (e.g., flow rate-limiting or flow quarantines), which can be flexibly managed by users/operators' high-level descriptions (see Section 3.4.2).

Advanced Sketch for Sampling. For scalabiliy issue, sampling is widely adopted in practice. For per-flow statistics, however, the estimation accuracy is ideal when exactly f/p packets for each flow are sampled, where f is the flow size and $1/p$ is the sampling rate. If more or fewer packets than f/p are sampled for a flow, it leads to over- or under-estimation of the actual flow size, because the number of the sampled packets is multiplied by p to estimate f. Therefore, the best strategy is to keep the per-flow sampling rate identical across flows. To that end, in the last phase of this dissertation, we aim to design the *per-flow systematic packet sampling*, which is a method to sample every p-th packet within a flow, whereas the well-known packet-level systematic sampling is to sample every p-th packet over the entire data stream. However, the complexity of the per-flow systematic sampling problem is equivalent to the per-flow counting problem, which means we still have to pay a large amount of memory/computations for the flow table (*i.e.* fail to reduce the complexity). To address this issue, we utilizes a sketch to reduce the complexity of the per-flow counting problem, for realize scalable counting.

1.3 Contributions

- (Wired domain) We design FlowRegulator to overcome the lack of scalability and online decoding capabilities. To verify its practicability, we further build a large-scale per-flow measurement system called InstaMeasure (Section 2.4).

- (Wireless domain) We proposed RFlow$^+$, a novel monitoring and management framework for WLAN, to support both *short-term* and *long-term* monitoring applications and enforce timely treatments (*i.e.* rate-limiting and flow quarantines) based on their requirements (*i.e. immediate* and *eventual*).

- (Wireless domain) We proposed a distributed packet counting algorithm for heavy user detection. The central collector holds a global view on the traffic measurement, while each on-site AP has its own local view for connected user measurement. The counting algorithm performs short-term measurement (*e.g.* 50 ms time window) locally as well as long-term measurement (*e.g.* 1 month) globally.

To verify and realize our proposed idea, we take following efforts:

- (Wired domain) To show InstaMeasure's feasibility and practicability, we implemented prototype of InstaMeasure using an off-the-shelf Atom processor board, and extended InstaMeasure to a multi-core measurement system (Section **??**).

- (Wired domain) We evaluate the performance of InstaMeasure in several scenarios. First, we evaluate the estimation accuracy and processing speed of InstaMeasure with 78 million L4 flows in one-hour CAIDA dataset by varying parameters (*e.g.* memory usage, the number of cores, packet per second, etc..). Second, we conduct a real-world campus network experiment for 113 hours by connecting InstaMeasure to a mirroring port of the main gateway router, capturing 9.11 billion packets, 122.3 million flows, and 8.5TB bytes. InstaMeasure successfully measured the whole L4 flows both in packets and in bytes where the standard errors of both estimations were smaller than 0.65% and. As one key application, InstaMeasure detected heavy hitters with 99.8% accuracy within 10 ms in the worst case—the prefix *Insta* comes from this tight time-bound (Section **??**).

- (Wireless domain) We prototyped RFlow$^+$ on top of OpenWrt on off-the-shelf access point hardware (TP-Link AC1750) as add-ons on OpenVSwitch (OVS) [73] and OpenDaylight [67]. The implemented RFlow$^+$ code is available upon request.

- (Wireless domain) To compare RFlow$^+$ with native OpenFlow and a sampling-based monitoring solution (sFlow), we performed real-world experiments by deploying our APs on our university campus.

1.4 Dissertation Organization

This dissertation consists of the contents from two papers [40–42]. Section 2 uses contents from Reference [42], coauthored with Seongkwang Moon, Youngtae Noh, Aziz Mohaisen and DaeHun Nyang, which proposes a sketch design to scale up per-flow design using large DRAM. Section 3 is based on Reference [40, 41], coauthored with DongGyu Cho, Aziz Mohaisen, Youngtae Noh, and DaeHun Nyang, which presents a two-layer counting mechanism that achieves immediate actions based on fast sketch and global actions based on sketch accumulations. Section 4 is based on Reference [**?**], coauthored with Daehong Ming, SeongKwang Moon, David Mohaisen, and Daehun Nyang, which proposes a new concept that per-flow systematic sampling to address the poor accuracy of the standard random sampling scheme. Some contents from these papers has been incorporated into introduction chapter of this dissertation.

2 Instant Per-flow Measurement Using Large In-DRAM Working Set of Active Flows

2.1 Introduction

We are inching closer to the zettabyte era with ever-increasing volumes of traffic on the Internet. According to Cisco's report [13], the annual Internet traffic will reach 3.3ZB per year by 2021. To deal with the rapidly surging demands on network bandwidth, per-port bandwidth now reaches 100 Gbps, or even more. To improve the utilization of the deployed network equipment (*e.g.* switch and router) by traffic engineering and secure networks, the role of traffic measurement becomes more important than ever, especially for data centers, where large volumes of traffic are moved between different sites or even with a single datacenter. Therefore, to enable fine-grained network traffic control, per-flow measurement (5-tuple: source IP address/port number, destination IP address/port number, and protocol) and its treatment become more crucial.

Thanks to the high-speed network traffic, measurement algorithms now have to cope with enormous incoming data rates (*i.e.* a larger number of flows) with tight deadlines (*i.e.* real-time). We stress that instant measurement is highly necessary for the data center traffic

engineering (TE) and network anomaly detection. For example, if denial of service (DoS) attack causes an influx of packets at 100 Gbps, the detection delay of 100 ms will cause 1.2GB data to hit a server or a network. Therefore, to eschew large bandwidth payment, instant anomaly detection is essential.

For per-flow measurement, sketch-based techniques have been greatly enhanced over several decades [15, 20, 24, 50, 54, 58, 78], starting with original proposals such as Flajolet-Martin (FM) sketch and Alon *et al.*'s approximate frequency measurement [1, 25]. Unlike their counterparts (*e.g.* NetFlow [12], sFlow [85], jFlow [43], etc.), sketch-based counting algorithms only require a small amount of memory to measure a large volume of traffic in real-time. To decrease memory usage, most works have used statistically shared counters [54], matrices [25], and Bloom filters [78] as statistical noise from each estimation can be removed at the time of estimation (or decoding). To enhance estimation accuracy, maximum likelihood estimation is usually adopted, thereby introducing a substantial amount of additional computations. Due to their designs, sketches are easily saturated when the number of tenant flows exceeds their capacity. Moreover, most of the sketch-based decoding algorithms involve hundreds of hash calculations (*i.e.* computationally overhead), and memory accesses from statistically mixed random blocks (*i.e.* latency overhead) [36] to obtain meaningful statistics (*e.g.* heavy hitters, DDoS attack, flow size distribution and entropy, etc.) [54, 58, 78]. For these reasons, sending the saturated sketch to a remote server for decoding is commonly accepted in practice but inherently incurs a huge network overhead and delay. For a software switch that is to be widely used in a data center server, remote decoding undoubtedly increases the network congestion, which degrades the user experience. Thus, online decoding is highly necessary for instant measurement and further timely detection.

To enable instant measurements, we can either decode the sketch on-site or use a local flow record table to perform the measurement. As discussed above, unfortunately, most sketch-based algorithms lack online decoding capability. For the local table, we naturally considered the working set of active flows table (hereafter, WSAF). A WSAF table is a flow record table that can be found usually in TCAM (Ternary Content Addressable Memory), CAM, or sometimes SRAM of a switching fabric for flow monitoring and management (*e.g.* switching, routing, or measurement). For instance, NetFlow uses TCAM and SRAM for storing WSAF in which an entry consists of a flow ID and the counting value [12]. In fact, the number of entries in the table cannot be large because those types of memories are quite expensive [18]. To scale up the WSAF table, we can put WSAF into DRAM instead of the expensive memory (*i.e.* the incentive to cost-effectiveness). However, there is a speed issue with In-DRAM WSAF: a packet arrival rate is too fast to handle by In-DRAM WSAF, owing to the DRAM's speed and WSAF table's hash collision.

Our approach is to put an online decodable sketch counting algorithm before the DRAM-

based WSAF table to slow down the incoming packet rate. Instead of directly inserting or updating every packet of flow into the DRAM-based WSAF table, we use a small sketch-based cache buffer, called FlowRegulator, to retain a fraction of flow counts. By doing so, we can suppress frequent accesses of the WSAF, thereby FlowRegulator can support the large-scale influx of flows with the slow-but-large DRAM. Consequently, FlowRegulator relaxes the need for precious memories (TCAM or SRAM) to maintain large WSAF, and further enables us to build a highly scalable and fast measurement system. Note that we originally built FlowRegulator with a state-of-the-art algorithm called Recyclable Counter with Confinement (RCC) [66] to enable the sketch's online decoding capability. However, we report a technical flaw of RCC and designed a better algorithm that can provide a more accurate estimation. Moreover, we propose a multi-layer sketch design to improve the scalability, which allows FlowRegulator to achieve a desired DRAM relaxation rate. In this paper, we deliver the following contributions:

- We propose a framework that uses a DRAM-based WSAF table to scale up the per-flow measurement capability. To compensate for the slow access speed of the DRAM, we suggest to put a FlowRegulator (a small flow buffer) before the WSAF table to relax the high flows influx rate.
- We report a technical flaw of a state-of-the-art sketch algorithm (*i.e.* RCC) and propose a new formula to improve the accuracy of the counting algorithm. Further, we realize the FlowRegulator with the improved algorithm, and design a multi-layer FlowRegulator to relax the flows influx rate at scale.
- We further build a large-scale per-flow measurement system called InstaMeasure, and prototype InstaMeasure using an off-the-shelf Atom processor board, and extended InstaMeasure to a multi-core measurement system.
- We evaluate the performance of InstaMeasure in several scenarios. First, we estimated the accuracy of the proposed algorithm with a one-hour real-world network trace (CAIDA) that contains 78 million L4 flows. Further, we compare FlowRegulator with a state-of-the-art algorithm (RCC). Second, we showed the efficiency of FlowRegulator in terms of decoding error, flow retention capacity, and flow relaxation rate. Third, and system-wise, we evaluate the packet processing speed and detection delay of InstaMeasure in a laboratory setting. Finally, we conduct a long-term real-world experiment by connecting InstaMeasure to our campus's main gateway router. InstaMeasure successfully measured the whole L4 flows both in packets and in bytes where the standard errors of both estimations were smaller than 0.65%. As one key application, InstaMeasure detected heavy hitters with 99.8% accuracy within 10 ms in the worst case—the prefix *Insta* comes from this tight time-bound.

2.2 Motivation

Unlike a small WSAF in TCAM and SRAM (*i.e.* industry practice), our DRAM-based WSAF table can store much more flows; thereby, we do not have to send flow records to a remote collector very frequently (*e.g.* every 10 ms). However, the downside is that we cannot evade the "sluggishness" of DRAM.

2.2.1 Managing WSAF at Packet Arrival Rate

DRAM's access speed is limited to process packets arriving at a line rate (*e.g.* 40 or 100 Gigabit Ethernet), so today's online measurement algorithms assume fast but expensive SRAM for processing sketches. Due to SRAM's prohibitive cost, only tens of megabytes are available to a counting algorithm [98]. Thus, instead of storing all the information of flows in SRAM, a measurement algorithm stores only a sketch or a summary in SRAM that does not have flow information (*i.e.* flow ID and its 5-tuple). A set of flow IDs in a table, a mapping between a sketch and a flow, or even a reversible sketch during a measurement period are normally stored in DRAM [83]. This use of DRAM is necessary and common in practice [54, 98], but managing flow IDs are quite challenging, and insertion-per-second (hereafter, ips) to the structure should be as high as packets-per-second (hereafter, pps). Also, in NetFlow, there exists a WSAF table in which ips should be high enough to process pps at a line rate. Under the constraint where {ips = pps} (insertion and lookup at WSAF should be done at packet arrival rate), it is hard for WSAF to keep up with the speed of the traffic increases. Packet sampling might be a viable option, which is used by NetFlow, SFlow, and many sketch-based schemes. However, such an approach degrades the estimation accuracy essentially. NetFlow uses both sampling and TCAM to ensure speed, but the most popular switching silicon chips have tables that can hold only up to thousands of route entries in TCAM and CAM [18], which cannot support a large-scale WSAF for instant measurement.

2.2.2 FlowRegulator to Relax {ips = pps} Constraint

Instead of using TCAM or SRAM, we can use DRAM for WSAF by relaxing the ips requirement for the WSAF table. Thus, instead of directly inserting or updating every flow packet into the table, we put a small buffer called FlowRegulator to retain a fraction of flow counts before WSAF. FlowRegulator has a memory block (or a virtual vector initialized to all 0's) for every single flow, and whenever a packet comes in, the corresponding block is updated by setting a random bit of the block. When the block saturates (or a portion of the block has set to 1's), the resulting counting fraction (we note that this is not the total size of flow) is

added up to the WSAF (*i.e.* a hash table in DRAM). Because FlowRegulator retains mouse flows whose sizes are smaller than the saturation condition, not all the packets are fed into WSAF, but only the packets that trigger the saturation condition are given to WSAF. This design greatly reduces ips even under a high pps condition.

2.2.3 How to Build FlowRegulator

To develop FlowRegulator, we utilize sketch-based counting algorithms, because they can encode packets at line rates, and can accurately estimate flows with a small amount of memory. Additionally, they satisfy our requirements: *online decoding* for adding up to WSAF when the block is saturated and *scalability* to deal with a large number of flows. A hitherto known solution is RCC proposed by Nyang and Shin [66] because it already has online decoding capability and proven to be useful for measurement in the wireless SDN environment [41]. To investigate its feasibility, we have tested RCC for its rate regulation (defined as Output ips/Input pps). Given that the access time of SRAM is 10-20 times faster than DRAM (and even faster with TCAM), RCC's rate regulation should be less than 5%. However, its regulation and retention capacity (the maximum number of packets in a virtual vector) are not operationally sufficient. To show that, we conducted an offline experiment using a CAIDA dataset [8]. As shown in Fig. 1, the solid line shows the actual packet arrival rate in pps, which is one mpps (million packets per second) on average, but RCC's saturation frequency is around 19% (output rate is about 190 kips (thousand ips) for the 8-bit vector, and 12% for 16-bit vector, which is far higher than the speed margin of SRAM over DRAM. Thus, we can conclude that RCC is not sufficient to build a scalable FlowRegulator. Even though we can have a better regulation rate by further increasing the size of the virtual vector, the improvement is insignificant. This will further be investigated in section 4.5.

Apart from the scalability issue, we report a logical error in the RCC's decoding and recycling processes, which leads to a biased flow estimation. Then, we provide a more reliable formula to explicitly estimate and eliminate mixed noises from the counter. This allows FlowRegulator to provide a more accurate estimation in the per-flow statistics and more explicit control in the flow regulation as well (See section 4.5.D). Moreover, we note that the RCC is not the only sketch that can be used to build FlowRegulator. One can use a better sketch to improve performance.

2.2.4 Two-layer Design for Higher Regulation Rate

Here, our observation is that enlarging the virtual vector size increases the retention capacity just in an addictive manner, and thus, this is not a viable (*i.e.* scalable) option. Instead, we

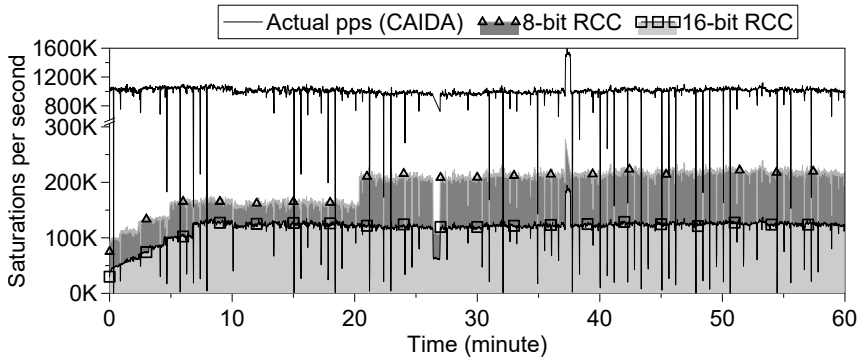


Figure 1: RCC's saturation occurs in the speed of 12-19% of packet arrival rate (the black solid line), which is too frequent to compensate for SRAM's speed margin over DRAM's (5-10%) in CAIDA dataset.

designed a new counting algorithm for FlowRegulator, which has two layers of probabilistic counters to achieve the higher rate regulation. Note that the concept of multi-layer sketch is not first introduced by this paper (*e.g.* [10]), but the only sketch-based data structure that supports online decoding. Our FlowRegulator plays a key role in retaining flows (from feeding into WSAF) for a while as well as counting flows. In the two-layer design, the second (higher) layer's one bit encodes multiple packets of a flow from a saturated sketch of the first (lower) layer. This design has substantially improved the rate regulation in a multiplicative manner. It enables higher rate regulation while not being detrimental to accuracy and speed while being scalable.

2.2.5 Saturation-based Decoding for Flows

Another aspect of FlowRegulator is counting elephant flows. Whenever a packet comes in a virtual vector, the estimation of the saturated vector is calculated by online decoding, and if saturated, the decoded counting value is finally accumulated to WSAF. This is called "saturation-based decoding" in contrast to "packet-arrival-based decoding". The latter is for actual online counting, and obviously, it is not feasible because of memory and computation speed. Saturation-based decoding has the property that it allows the only elephant flows (flow sizes greater than retention capacities of the sketch) get through FlowRegulator to reach the WSAF table, which prevents WSAF from exploding from a huge number of incoming mouse flows. This is in contrast to NetFlow, which registers every flow, if not sampled, in the table regardless of its size. Owing to this, WSAF can keep the counters only for active elephant flows, which means FlowRegulator helps to maintain a WSAF with good quality. Notably, even though our FlowRegulator filters mouse flows well, there are still mouse flows

that get through to WSAF (recall that FlowRegulator is a probabilistic counter). We note, however, that it is essential for some applications to have samples of mouse flows (*e.g.* DDoS attack, SuperSpreader, and entropy etc.). However, WSAF needs to evict the expired (or least significant) mouse flows when the table is full.

For FlowRegulator, instead of running a separate core periodically (NetFlow approach), when a new flow is inserted, and an empty slot is searched by hash chaining, garbage collection is performed. Using our WSAF in DRAM, we can also analyze flow behavior for long-term measurement. Considering that other sketch-based schemes send a sketch and flow ID information periodically to a remote collector for sketch decoding, the decoding can be regarded as a "delegation-based decoding". Comparing the three different approaches, namely the delegation-based, the packet-arrival-based (used as ground truth and a baseline), and the saturation-based decoding, we note that the packet-arrival decoding has the fastest detection time. However, the time difference between packet-arrival-based and the saturation-based decoding is within 10 ms, while the difference between packet-arrival-based and delegation-based decoding is tens of milliseconds (may increase depending on network delay). Therefore, our saturation-based decoding is substantially faster than delegation-based decoding.

2.3 Related Work

There are two major challenges when implementing DRAM-based WSAF table: hardware constraints and computational complexity. First, off-chip DRAM is well known for its high delivery latency of the first word that a CPU requests (roughly 10 ns). Assuming 40 Gbps traffic, only 12 ns for per-packet processing is available for 64 Bytes packets, which is almost impossible to be accomplished considering the necessary tasks (*i.e.* 5-tuple extraction, hashing, probing and counter update). Second, even the fast SRAM (say, one ns access latency) cannot help, because the hash table itself requires a huge computational overhead due to hash collisions. This is why the on-chip and collision-free TCAM (*i.e.* NetFlow) is the only feasible solution to maintain WSAF at the line speed. The alternative solutions to TCAM can be categorized into three: sampling, sketch, and selective monitoring.

Sampling Approach. Sampling technology is widely used in practice because of its simplicity [85]. Generally, it randomly samples only one packet among every k packets using a simple trigger located in the packet processing pipeline. From the perspective of WSAF, sampling relaxes the influx rate to the hash table from 1 to $1/k$, thereby more CPU cycles are available for flow record the insert, update, and delete operations. However, the major drawback of this approach is the poor trade-off between the sampling rate and accuracy, meaning that lowering the sampling rate degrades the estimation accuracy essentially.

Moreover, the sampler randomly chooses samples among the entire traffic, which fails to provide fine-grained samples and in turn leads to the inaccurate estimation of fine-grained flows (*e.g.* layer-4 flow). The other shortcoming of this approach is the incomplete statistics of the mouse flows. Kumar *et al.* suggested using a non-linear approach to perform sampling that samples less from elephant flows, and more from the mouse flows [51]. Later, Hu *et al.* proposed a similar idea that uses the non-linearity of sampling to achieve better accuracy in mouse flows. These approaches achieved more complete statistics, but they failed to provide a stable sampling rate that is crucial to online performance.

Sketch Approaches. A large volume of works on sketch-based measurement have been done to leverage its estimation accuracy for traffic engineering and anomaly detection [15,17, 20,50,54,58,66,78,83,96]. Notable works on real-time measurement systems include OpenSketch, which utilized various sketches and specialized hardware: TCAM and SRAM [98]. FlowRadar takes advantage of a recently proposed hash data structure called the Invertible Bloom Lookup Table (IBLT) to resolve the hash collision problem [29,55], and UnivMon [57], which uses a method named universal streaming [6]. Especially, the view of FlowRadar on WSAF is similar to InstaMeasure, although it tried to solve non-deterministic insertion time by the constant time insertion of IBLT, and delegate the decoding process to a remote server, which presents a huge network bandwidth overhead. Application-wise, Estan and Varghese's work was on heavy-hitter detection during a measurement period [24], which was followed by several other works [16,20,36,44,46,48,83]. Recently, Basat *et al.* proposed an elephant flow identification and a top-k counting algorithm [2,3]. Their top-k is quite limited (up to top-512). InstaMeasure is concerned with the larger scale of top-k, *e.g.* tens of thousands to millions.

Selective Monitoring. Another way to reduce the hash table's burden is using selective monitoring, which ignores a portion of the flows to guarantee online performance. Trumpet [63] is a host-side approach that maintains flow records in the DRAM's hash table. To be resilient to DDoS attacks, Trumpet adopts a filter table to discard flows less than a threshold, which is calculated offline according to the processing speed of the Trumpet module. Elastic sketch [94] also maintains a hash-based flow record table. Instead of discarding the mouse flows, Elastic sketch utilizes a fast probabilistic data structure Count-Min sketch [17] to minimize the overhead.

Among these approaches, InstaMeasure has a similar view on WSAF to FlowRadar but does not delegate the measurement to a remote server. In terms of overhead minimization, InstaMeasure uses a compact data structure (*i.e.* sketch) to realize the real-time performance. At the same time, the sketch plays an important role that relaxes the influx of the WSAF table to compensate for the slow access of the DRAM memory.

2.4 FlowRegulator Design

Today's Internet traffic follows a Zipf-like distribution [7], and mouse flows (*e.g.* 1-10 packets flows) are the majority of network flows, which is the main reason for WSAF cache saturation. The DRAM is relatively cheap; thus, we have fewer constraints on its use, compared to SRAM and TCAM. To overcome its slow read/write access time, we designed a sketch-based FlowRegulator to regulate influx rates of packets in front of WSAF by retaining mouse flows until they overflow (or saturate) sketches that they reside in. Note that most mouse flows do not grow enough to overflow their sketches.

2.4.1 Architecture and High-level Design

Fig. 20 illustrates our design of FlowRegulator. The L1 counter is a sketch-based data structure introduced in RCC (Recyclable counter with confinement [66]). The authors of RCC proved that a small virtual vector (8-bit) provides a higher estimation accuracy. A major problem, however, is that if we use RCC for FlowRegulator, the 8-bit virtual vector can only count up to tens of packets in the best case. That means the structure can retain only a small portion for each flow. Once the vector is saturated, the flow count must be accumulated in the WSAF table, and then the vector is recycled for the next round of counting. Therefore, the small flow retention capacity leads to frequent insertion operations of the WSAF table, which is not acceptable for In-DRAM WSAF: Fig. 1 of RCC's flow regulation rates for two vector sizes shows the vector size increment, which does not effectively increase the regulation rate. To address this problem, we use a two-layer sketch strategy to increase FlowRegulator's retention capacity significantly by designing the second layer sketch to count in multiple units of the first layer sketch. This multiplicative approach allows FlowRegulator to retain larger mice and to retain more packets of each elephant flow (up to around 100 packets for a single flow—10 times more than that of RCC).

As shown in Fig. 20, the L2 counter is a set of L1 counters. We categorized L1's estimation into four cases based on the surrounding noise level. Then, we use those four different estimation values (*e.g.* $1-4$) as the units of four counters in the second layer. For example, when a virtual vector is saturated in L1 and the estimated value is 4 (among $1-4$), the fourth L2 counter is chosen to perform the same counting task as L1. If the estimated value of L2's fourth counter is 3, the total counting value would be 12 ($=4\times3$). The encoding and decoding processes of L2 counters are designed to be the same as that of L1, and even the memory layout and the virtual vector's bit positions of every flow are the same (hash function reuse of L1 virtual vector). Thus, L2 counting only requires one additional memory access (in total, two memory accesses, and one hash, including L1 counting). By doing this, we obtained around 1.02% flow regulation rate; thus, the insertion request rate to the WSAF

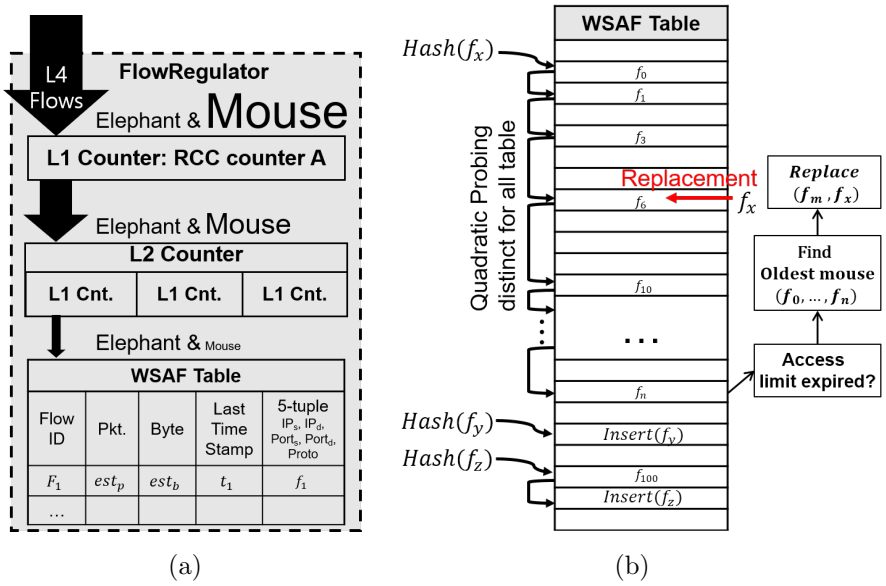


Figure 2: Design of FlowRegulator: (a) Components of FlowRegulator (b) Probing limit-based second-chance replacement policy of WSAF Table

table could be reduced substantially (See section 4.5).

2.4.2 Encoding and Decoding

In the following, we illustrate how FlowRegulator estimates the flow size. FlowRegulator has the same encoding process as RCC but uses different decoding strategies to improve the accuracy and to realize a multi-layer design. For better understanding, we describe FlowRegulator's design according to RCC's notation, as shown in Table 1.

Given a s-bit bitmap (s) and incoming packets from a flow set $f = \{f_1, f_2, ..., f_n\}$, linear counting (LC) [93] is used to perform cardinality counting by sequentially flipping $h(f_x)$'s position to 1. The number of flows is estimated as $\hat{n_f} = -s\ ln(V_s)$, where V_s is the fraction of 0's bit after encoding all packets. Introduced by Nyang *et al.* [66], RCC used a randomization technique to use LC's theory for multiplicity counting. That is, for n incoming packets from a single flow f, RCC flips one "randomly-chosen bit position" to 1 for each incoming packet. Following the theory of LC, the number of packets is estimated as $\hat{n} = -s\ ln(V_s)$. Further, Nyang *et al.* improved the accuracy of LC's estimation by using non-approximation formula:

$$\hat{n} = \frac{\ln V_s}{\ln\left(1 - 1/s\right)}, \tag{1}$$

where s is the size of the bitmap and V_s is the fraction of 0's bits after encoding (See [66] for the detailed derivation).

Table 1: Notation

$h()$	hash function	v	virtual vector
f	flow ID	$\hat{n_f}$	number of flows
n	number of packets	$\hat{n}$	estimated number of packets
$\hat{noise}$	estimated noise	$\hat{k}$	estimation after eliminating noises
s	virtual vector size	V_s	the fraction of 0's in v
m	entire bitmap size	V_m	the fraction of 0's in m
w	word size	V_w	the fraction of 0's in w
z	# of recycled bits	K	set of $\hat{k}$ for different zs

Inspired by the compact spread estimator (CSE) [98], RCC applied the concept of virtual vector to realize multi-set multiplicity counting. The virtual vector is a bitmap (memory block) that is designed to share bit positions with other bitmaps. The main purpose of sharing is to minimize memory usage, leading to noise in the virtual vectors as a trade-off.

Encode. For each incoming flow f, RCC assigns a s-bit virtual vector among a large m-bit array. Then, performing multiplicity counting using the randomization technique. To deal with speed issues, RCC suggests using a small virtual vector s and confines the vector within a word (32 or 64 bits) to consume only one memory access for encoding/decoding. The small-sized virtual vector leads to frequent saturation of the vector: thus, RCC accumulates the estimation vector to a hash table and recycles the vector for the next round estimation.

Decode. As mentioned above, the virtual vector contains noises for sharing of bit positions among multiple flows. Based on CSE, RCC eliminates the noise from the estimation $\hat{n}$ as

$$\hat{k} = \hat{n} - \hat{noise} = \frac{\ln V_s}{\ln(1 - 1/s)} - \frac{s \cdot \ln V_m}{m \cdot \ln(1 - 1/m)}, \tag{2}$$

where m is the size of the entire bitmap and V_m is the fraction of 0's of the bitmap. The second term of $\hat{k}$ is the estimation of the entire bitmap divided by the number of non-redundant vectors, which is considered as the amount of noise in RCC.

In FlowRegulator, we can start the noise estimation with

$$\hat{noise} = \frac{s \cdot \ln V_w}{w \cdot \ln(1 - 1/w)}, \tag{3}$$

where w is the size of the word, and V_w is the fraction of 0's among the word to which the virtual vector belongs. Unlike RCC, we consider only the noise coming from the flows in a word instead of the entire memory space because RCC confines a virtual vector to be

distributed only within a word, which means the noise in a virtual vector is donated only by the other virtual vectors that are located in the same word, but not vectors outside of the word. Furthermore, considering the estimated virtual vector (flow) is mixed with counts and noise, we exclude the virtual vector from the noise estimation as

$$\hat{k} = \hat{n} - \hat{noise} = \hat{n} - \frac{s \cdot \ln V_{w-s}}{(w-s) \cdot \ln(1 - 1/(w-s))}, \tag{4}$$

which estimates the noise based on the bit positions in a word but exclusive of the virtual vector itself.

Recycle of Virtual Vector. FlowRegulator uses a small virtual vector for better accuracy but also faces the vector saturation problem. The recycling event of a virtual vector is triggered while decoding when a virtual vector reaches its counting limit (*i.e.* 70% of bit positions flip to 1's [93]). The recycling process of FlowRegulator is to restore 1's bits to 0 until the portion of 0's in v (V_s) is equivalent to the portion of 0's in v except bit positions in the same w (V_{w-s}). Thus, the number of 1's bit that has to be restored to 0 is

$$z = V_{w-s} \cdot s - V_s \cdot s, \tag{5}$$

where the former part ($V_{w-s} \cdot s$) is the target number of 0's bits of v after recycling, and the later part ($V_s \cdot s$) is the current number of 0's. Because V_s in the second term is a fixed value once s is decided, z is dependent only upon V_{w-s}. Furthermore, $\hat{k}$ also depends on V_{w-s} according to formula (4). However, we note that since the number of 1's bits to be recycled (z) must be an integer ranging from 1 to $0.7 \cdot s$, z has to be approximated first to calculate the corresponding $\hat{k}$. To this end, per different z's, we have different estimations (*i.e.* $K = \{\hat{k_1}, \hat{k_2}, \hat{k_3}, ..., \hat{k_z}\}$). For the single-layer design of FlowRegulator, one of the estimations in K accumulates to the WSAF table along with the flow ID that triggered the recycling event. For multi-layer design, we use $|K|$ numbers of the L1 counter (*i.e.* word array) to record the number of recycling events that occurred with each $\hat{k_z}$, separately, as shown in Fig. 20. For instance, L2 is a collection of L1 counters, where the first L1 counter (*i.e.* L2[1][]) is responsible for counting a flow that L1's estimation is $\hat{k_1}$ (*i.e.* one 1's bit is recycled after saturation), and the rest of L1 counters follow in the same manner. Since the L2 counters follow the same mechanism of L1 counter, the saturation recycling event eventually occurs in L2 counters. Upon saturation at L2[1][], the final estimation is $\hat{k_1} \cdot \hat{k_z}$, where $\hat{k_1}$ is the estimation at L1, and $\hat{k_z}$ is the estimation at L2.

2.4.3 WSAF Table Management

Our FlowRegulator can retain most mouse flows, but not all of them. There still is a probability for mouse flows to pass through FlowRegulator and to be inserted into the

Algorithm 1: Two-layer FlowRegulator

```
1  Init L1[ ];
2  Init L2[Noise_min][ ], ...,L2[Noise_max][ ];
3  forall Pkt_f do
4      (idx_f, v_f) ← Hash(Pkt_f);
5      Noise_L1 ← RCC_Encode(L1[idx_f], v_f);
6      ;
7      if Noise_L1 ≠ NULL then
8          /*v_f saturated in L1*/;
9          Noise_L2 ← RCC_Encode(L2[Noise_L1][idx_f], v_f);
10         ;
11         if Noise_L2 ≠ NULL then
12             /*v_f saturated in L2*/;
13             unit ← RCC_Decode(Noise_L1);
14             est_pkt ← unit × RCC_Decode(Noise_L2);
15             est_byte ← est_pkt × Length(Pkt_f);
16             ACC_WSAF(f, est_pkt, est_byte)
17         end
18     end
19 end
```

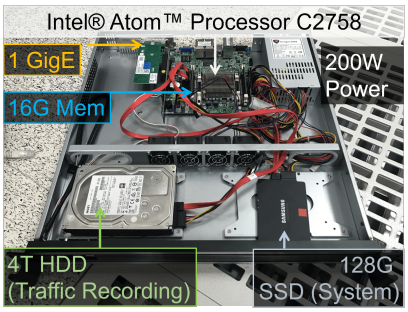


Figure 3: InstaMeasure as a measurement device

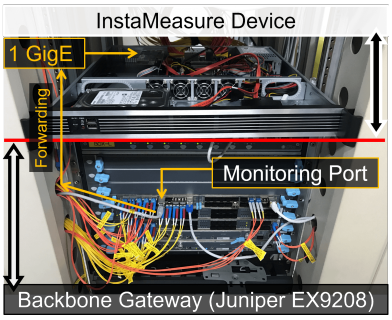


Figure 4: Configuration of real-world experiment

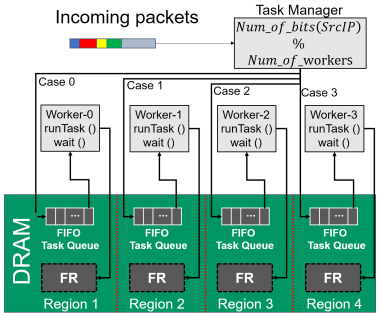


Figure 5: Multi-core flow regulation

WSAF table owing to noise. These mouse flows lead to memory space wastes and frequent hash collisions (*i.e.* probing of active flows increases). We address this problem by using a probe limit-based and second-chance replacement algorithm to evict mouse flows from the WSAF table to save memory space and increase probing speed. Moreover, the probe limit-based approach allows us to use specific parameters (*i.e.* table size $m = 2^n, h(k,i) = hash(k) + 0.5i + 0.5i^2 \ mod \ m$) for probing all table positions in $[0, m-1]$ to achieve a high load factor. See Fig. 2(b).

2.4.4 Sampling-based Byte Counter

InstaMeasure has another desirable feature that provides packet and byte counting at the same time. Based on the packet counting technique, we utilize a sampling-based approach to perform byte estimation. When a flow f saturates FlowRegulator, an estimated packet number (est) will be accumulated to the WSAF table using the f_{id}. We use the size of the last arrived packet len to multiply with est and accumulate $len \times est$ to the byte counting field of WSAF table. Even though the idea is straightforward, it works quite accurately ($< 1\%$ error rate, see section 4.5) and efficiently (one extra multiplication).

2.4.5 Algorithm

L1 counter of FlowRegulator has a simple word array structure, where the size of each word is selectable (32 or 64 bits depending on processor). When a packet arrives from flow f, FlowRegulator computes a hash function using 5-tuple extracted from the packet (line 4). The hash value is used for two purposes, 1) to extract virtual vector v_f (*i.e.* bit positions confined in a word—virtual vector confinement technique as in [66]), and 2) to determine v_f's

word location (idx_f) at L1 counter (L1[idx_f]). Once idx_f and v_f are decided, `RCC_Encode` performs encoding of the sketch until v_f of L1[idx_f] saturates and returns a noise level ($Noise_{L1}$) (line 7). L2 is a set of L1 counters. When the saturation happens in L1, one of the counters in L2 will be selected depending on $Noise_{L1}$ to perform second layer counting using the same idx_f and v_f (line 9). When v_f is saturated in L2, FlowRegulator estimates the total packet number (est_{pkt}) by multiplying `RCC_Decode`($Noise_{L1}$) and `RCC_Decode`($Noise_{L2}$), where the former is the number of packets at L1 at the saturation moment, and the latter is the frequency of saturation at L2 (lines 14-15). The estimation of byte volume (est_{byte}) is done by the saturation-based sampling approach. That is, the byte volume is calculated by multiplying est_{pkt} with the size of the packet that triggered the L2 saturation (line 15). Finally, FlowRegulator accumulates est_{pkt} and est_{byte} to the WSAF table using flow ID f (line 16) either by insertion or by update.

2.5 Implementation

We prototyped InstaMeasure in an off-the-shelf device with 8-Core Atom processors. The estimation accuracy and the processing speed of InstaMeasure were evaluated by a packet-driven experiment using a one-hour CAIDA dataset (1-4 cores used). Further, we set up a real-world experiment using InstaMeasure device at the backbone gateway router of our campus network for 113 hours autonomously and ran a use case: heavy hitter detection (1 core used).

2.5.1 Hardware Description

Fig. 3 shows the hardware setup of our InstaMeasure device. We used a Supermicro motherboard A1SRi-2758F that embeds 8-Core Intel Atom processor C2758 ($312), which has a 4MB cache memory (448KB for L1 cache and 4096KB for L2 cache). In total, 16G (2x8G) DDR3 1600MHz memory was used with a 200W power supply. We used a 128G SSD for running Linux 16.04 server (x86) and 4T HDD to record the network trace for offline analysis. For fast packet processing, we implemented InstaMeasure based on DPDK (version 17.11.2) to bypass the kernel. Note that our choice of the CPU is reasonable as Atom series CPU appears in many modern routers/switches, including bare metal switches [65].

2.5.2 Real-world Experiment Setup

Our campus uses 2 Gbps bandwidth in total (1 Gbps for up-link and 1 Gbps for downlink), and the backbone gateway router uses a Juniper EX9208 switch, as shown in Fig. 4. Since,

for logistical reasons, the gateway could not be programmed for this experiment, we used the mirroring port of the gateway to perform our measurement. The purpose of this experiment is to check InstaMeasure's performance (CPU and memory use) and scalability (accuracy for 113 hours) (See section 4.5.D for results). We also ran a use case of heavy hitter detection. Because the mirroring port starts to drop packets when port capacity is exceeded, the estimation accuracy was evaluated by comparing results of InstaMeasure to results obtained by the recorded traffic experiencing the same packet drop. Due to the policy of our school, we were permitted to access only the up-link, although for a long time. Moreover, we evaluated the processing speed and heavy hitter detection delay using the CAIDA dataset and artificially-generated traffic, to cope with non-deterministic mirroring delays caused by port buffering in our real-world experiment.

2.5.3 Multi-core Traffic Measurement System

To perform faster encoding and decoding by taking advantage of the multi-core Atom processor, we implemented InstaMeasure as a multi-core traffic measurement system. Fig. 5 shows a case of the four-core model. As shown, we allocate memory blocks exclusively to each worker core to avoid memory collision, where each worker core maintains an independent FlowRegulator structure with a FIFO task/packet queue. A worker continuously monitors its task queue and performs encoding and (if necessary) decoding whenever each packet arrives. An additional manager core is responsible for allocating packets to a worker's queue. To evenly distribute packets to be processed, the number of 1 bit of source IP address is used to determine which queue the packet goes into. As will be shown in section 4.5.C, InstaMeasure scales based on the number of core.

2.5.4 Parameters

The main component of FlowRegulator is the two-layer counter. To construct FlowRegulator, we used a total of five small counters, one for L1 and four for L2, as described in section 2.4. Thus, when we use M memory space for the two-layer FlowRegulator, five small counters equally assigned with $M/5$ memory. Moreover, in the multi-core system, the total memory usage is M times the number of worker cores. Thus, for the four-core system, the allocated memory is $M \times 4$.

In a lab experiment, we evaluated the accuracy of a single core FlowRegulator using the CAIDA dataset by varying the memory usage of the FlowRegulator from 128KB to 2048KB. In the real-world experiment, we used 128KB of memory with a single core worker. FlowRegulator's processing speed was shown to be fast enough to process 10 Gbps links

(see section 4.5). For the memory usage of the WSAF hash table, we fixed the total entry numbers to 2^{20} for all experiments, including the multi-core case. As shown in Fig. 20, the size of each hash table entry is 33 bytes to include a flow ID (32 bit hash of 5-tuple), packet counter (32 bits), byte counter (32 bits), timestamp (64 bits) and the 5-tuple (104 bits). Thus, the total DRAM space required for the hash table is only 33MB. If we allocate more DRAM, *e.g.* 1GB, it can run for several days autonomously and without interruptions on a 10 Gbps link.

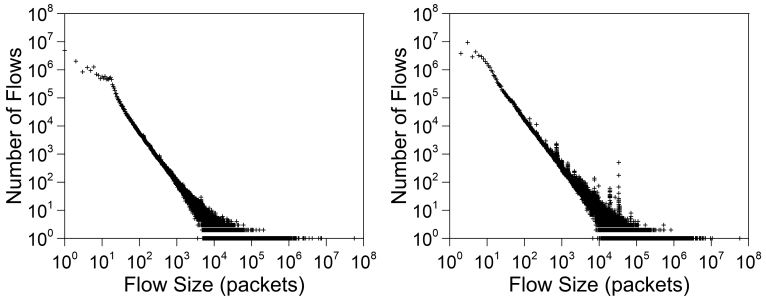
2.6 Evaluation

In this section, we show the feasibility of our InstaMeasure system through various experiments. First, we show the estimation accuracy and flow relaxation performance of FlowRegulator using a one-hour CAIDA dataset. Second, we discuss the trade-off of FlowRegulator using various experiments. Third, we show the overhead of the InstaMeasure system, such as processing speed, memory usage, and detection delay. Last, we demonstrate a real-world experiment to verify the feasibility of InstaMeasure.

2.6.1 Datasets

- **CAIDA Anonymized Internet Trace 2016.** [8] We used one-hour (13:00-14:00, 6th of April, 2016) network traffic trace that was collected at the Equinix-Chicago data center on an OC-192 link (maximum load of 10 Gbps). We merged trace data of both directions (*i.e.* between Chicago and Seattle) in the order of timestamp to evaluate InstaMeasure with larger-scale network trace. As a result, our dataset contains 3.7 billion IPv4 packets (including UDP, TCP, and ICMP), 78 million L4 flows, and the highest speed was 1.5 mpps (million pps). This scale is substantially large and beyond the current sketch-based measurement's capability. See Fig. 6(a) for the traffic distribution of the dataset.

- **113-hour backbone gateway traffic on-campus network.** We implemented our InstaMeasure in an off-the-shelf device and measured up-link traffics (1 Gbps bandwidth) at the backbone gateway (Juniper EX9208 switch) of our campus for 113 hours in total. For further analysis, we also recorded 5-tuple, the packet size, and the timestamp of every single packet. In total, about 8.5TB of traffic, 9.1 billion packets (broken down into 6.4% of UDP and 93.6% TCP), and 122.3 billion L4 flows were observed in 113 hours. See Fig. 6(b) for distribution.



(a) CAIDA (b) Backbone gateway of Campus

Figure 6: Distribution of CAIDA dataset and 113 hours campus traffic.

2.6.2 Metrics

We use the following metrics to evaluate InstaMeasure.

- **Relative Error**: $(\hat{f} - f)/f$, where $f(\hat{f})$ is the actual (estimated) flow size. We use RE to show the estimation error of the fine-grained flows.
- **Average Relative Error**: $ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|f_i - \hat{f}_i|}{f_i}$, where n is the total number of flows. ARE presents the overall accuracy of the flow size estimation. Due to the heavy tail distribution, inaccuracy in estimating mouse flows leads to a large ARE.
- **Recall of top-k**: $TP/(TP+FN)$, where TP is the number of recorded flows of which size is equal to or greater than the K-th flow, where the size of K-th flow is from the ground truth. FP is the number of recorded flows which the size is smaller than the K-th flow, and FN is the number of non-recorded flows of which the size is equal to or greater than the K-th flow. We use the recall to evaluate the quality of our top-k list.

2.6.3 Evaluation of FlowRegulator

WSAF ips Relaxation. In Fig. 7, the x-axis represents the timeline of our merged CAIDA dataset, and the solid black line on the top represents the actual pps of the trace. Below the pps line, RCC's and FlowRegulator's regulation rates are shown in red squares and blue diamonds, respectively. The figure shows that RCC relaxes ips to feed packets to the WSAF table at the speed of 112 kips (thousand ips), which corresponds to a 12% regulation rate. FlowRegulator effectively regulated flows to pass only 1.02% with 128KB DRAM memory, Considering that WSAF is usually stored in SRAM or TCAM, and SRAM is 10-20 times

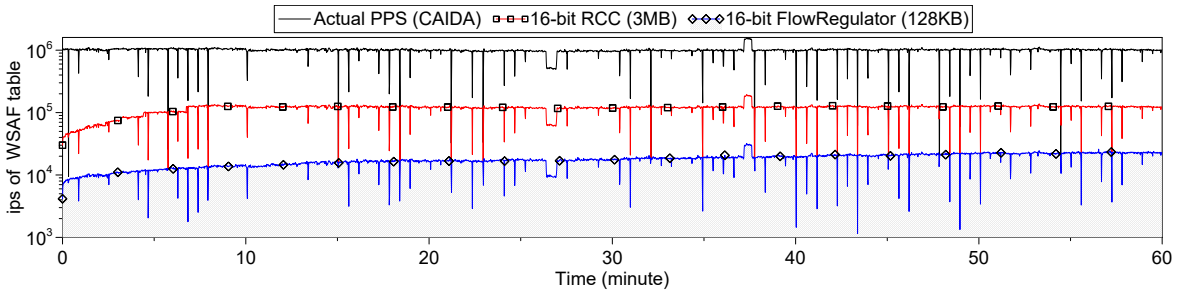
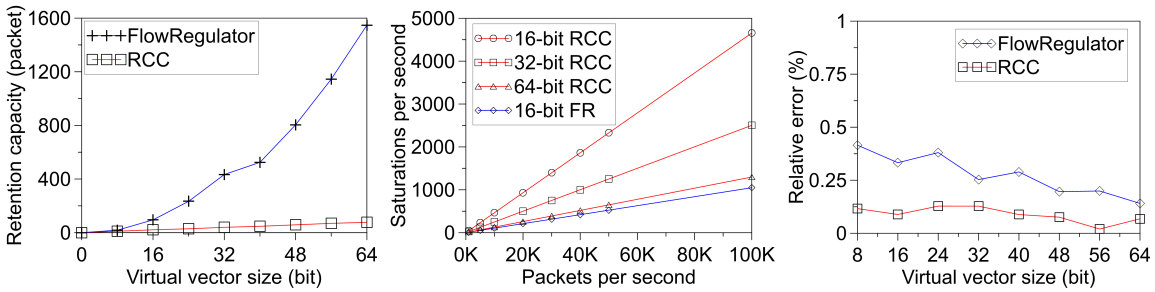


Figure 7: WSAF relaxation: FlowRegulator (FR) and RCC ips of CAIDA dataset



(a) Flow retention capacity (b) Saturation frequency (c) Decode error per saturation

Figure 8: FlowRegulator's retention capacity and saturation frequency outperforms RCC's, paying a little degradation of accuracy.

faster than DRAM, FlowRegulator has a sufficient margin, while RCC does not have as can be seen in Fig. 7. Even for WSAF in TCAM, which is faster than SRAM, FlowRegulator can be configured to have enough margin by adjusting the vector size or even the number of layers.

Regulation Rate vs. Sketch Size. Since FlowRegulator's role is to slow down the insertion request rate to WSAF, we evaluate how effectively it achieved this goal. Fig. 8(a) shows comparatively the retention capacity of each virtual vector by varying its size. For RCC, the growth rate of the retention capacity is very slow; thus, its retention capacity is only 77 packets even with a 64-bit virtual vector. Note that to use a 64-bit virtual vector, the confinement size should be at least 256 bits, which incurs eight memory accesses and eight hash computations for every packet in a 32-bit system, which is not acceptable for FlowRegulator. Compared to RCC, FlowRegulator's retention capacity grows very quickly as the size increases, and thus a 16-bit vector (8 bits for each layer) is enough to retain a hundred flows. To fairly compare FlowRegulator of two layers to RCC of a single layer, FlowRegulator's vector size is defined to include all the vectors where a packet can reside—since we

are interested in the number of packets retained by a virtual vector. Since FlowRegulator's design has two layers, it would be twice of L1 counter's virtual vector size. Fig. 8(b) shows the saturation frequency of a sketch for a single flow comparatively, which indicates that the insertion request rate to WSAF is decreased (better for WSAF) as the frequency becomes low. The figure shows that RCC with a 64-bit virtual vector seems to be barely comparable to FlowRegulator, but it is impractical, as we mentioned above. Also, in the real world, a sketch accommodates a large number of flows, so the saturation rate is much higher than that in the analysis, as shown in Fig. 7. Thus, even a larger vector for RCC should be utilized. Consequently, as shown in Fig. 7, FlowRegulator provides enough retention capacity to suppress the insertion request frequency, which cannot be achieved by RCC.

On Cost. Two-layer design of FlowRegulator, however, pays a small penalty of accuracy degradation, which is shown in Fig. 8(c). The overall accuracy of FlowRegulator is lower than that of RCC with a single layer, but the difference is very small except when the vector size is 8 bits (4 bits for each layer). We note that FlowRegulator implementation for all the experiments has a 16-bit long vector. Another cost might be the detection latency: because FlowRegulator relies on sketch saturation-based decoding, an event such as heavy hitter cannot be detected immediately, but when the flow is registered in the WSAF table. This, in turn, delays the detection. However, as shown in Fig. 14(b), the delay is less than ten milliseconds, which is negligible compared to tens of milliseconds of delay in most frameworks (*e.g.* [55]). Also, in the same figure, we draw that significant attackers use more bandwidth, and thus can be caught earlier than slow attackers, who are less important in volume-based attacks.

2.6.4 Accuracy

FlowRegulator vs. RCC. For a fair comparison, we use 0.5MB memory for RCC, single-layer FlowRegulator and two-layer FlowRegulator. Fig. 9−11 show the experiment results of each algorithm. In each figure, the most left one is the overall view of the estimation accuracy in log scale, and the rests (b-c) are the estimation results of different flow sizes in linear scale, ranging from 0 to 1,000, 1,000 to 10,000, respectively. Again, the last one is in log scale for flows that are sized from 10^4 to 10^7. The x-axis represents the actual flow size, and the y-axis is the estimated flow size. A red guideline $Y = X$ helps to show the estimation accuracy. Each point stands for one flow, the data points that are under the guideline mean under-estimation of flows, and otherwise means an over-estimation of flows. Thus, the closer the data points are to the guideline, the more accurate they are.

As shown in Fig 9(a), RCC seems to be accurate on a log scale. However, under the linear scale, we found that the estimation of RCC is biased for larger flows. As described

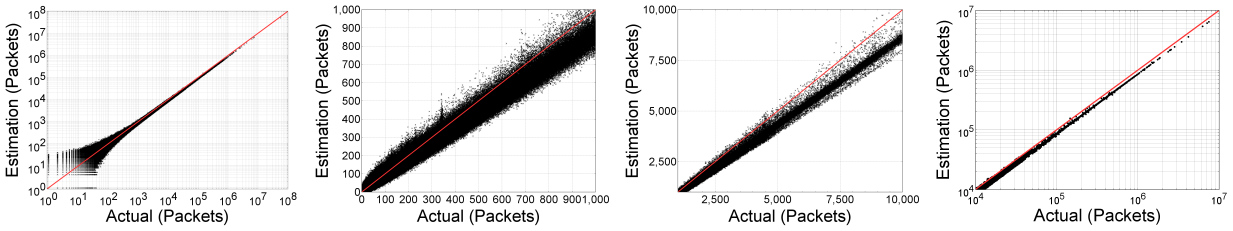


Figure 9: Recyclable Counter with Confinement (RCC): Estimation results for RCC with 0.5 MB memory. Each data point stands for each flow, and the line $Y = X$ is the guideline. To see how accurate each algorithm is, check how close every point to the guideline. (a) Overall estimation results in log scale (b) Estimation in linear scale from 1 to 10k. (c) Estimation in linear scale from 1k to 10k. (d) Estimation in log scale from 10K to 10M.

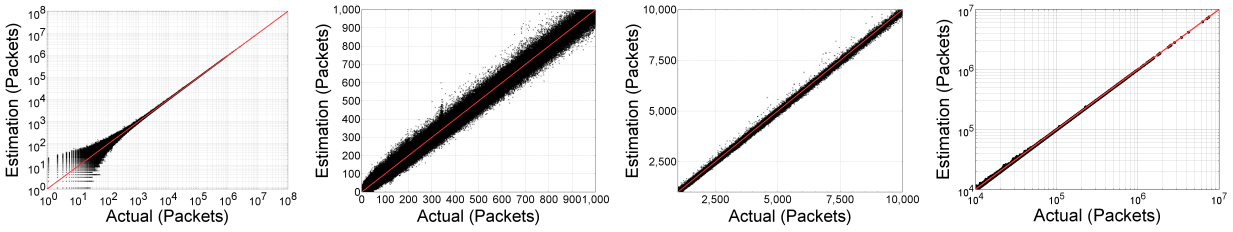


Figure 10: Single-layer FlowRegulator: Estimation results for single-layer FlowRegulator with 0.5 MB memory.

in section 2.4.B, RCC calculates the noise after each virtual vector saturation; however, the calculation of noise within a vector is based on the average noise of the entire memory space, which is irrelevant to the vector that is confined in a single word, which leads to an inaccurate estimation. Moreover, since RCC uses a small-sized virtual vector, the frequent virtual vector saturation leads to an accumulation of errors, which presents large errors in the elephant flows. On the other hand, FlowRegulator considers and calculates noise only within a word that hosts the virtual vector, thus the estimation of FlowRegulator is more accurate than that of RCC, as shown in Fig. 10. Furthermore, as shown in Fig. 11, FlowRegulator with two layers shows a bigger estimation variance than with a single-layer, which is verified the result in Fig. 8(c). This is the cost of having a scalable counter. However, the overall estimation is unbiased, and accuracy degradation is small.

Estimation Variance of Elephant Flow Over Time. For robust anomaly detection, estimation of every single flow is required to be accurate at any moment. In this experiment, we show the relative error of giant flows (having more than 10^7 packets) every second.

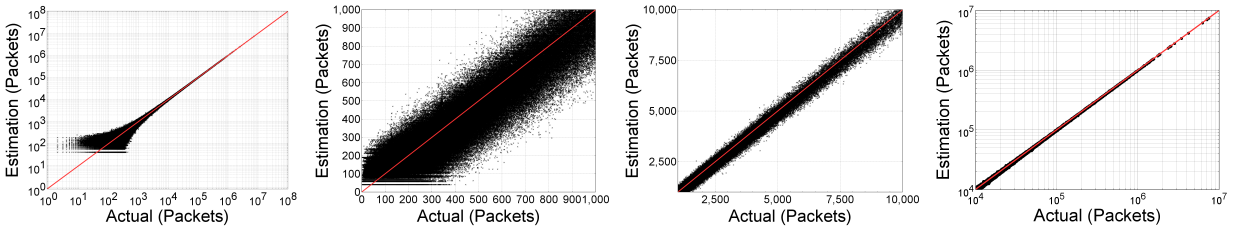


Figure 11: Two-layer FlowRegulator: Estimation results for two-layer FlowRegulator with 0.5 MB memory.

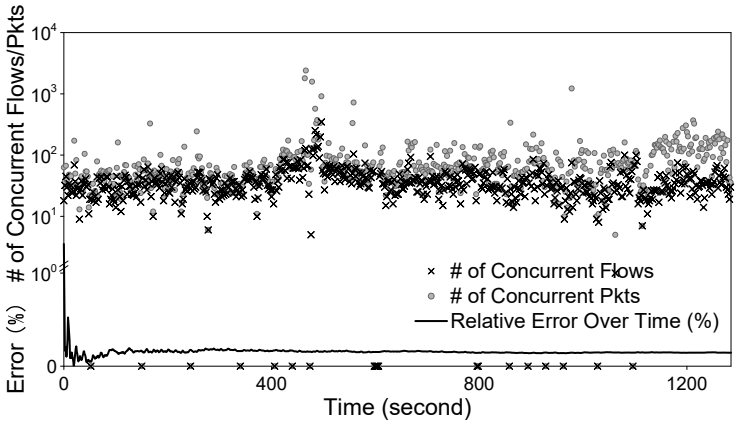
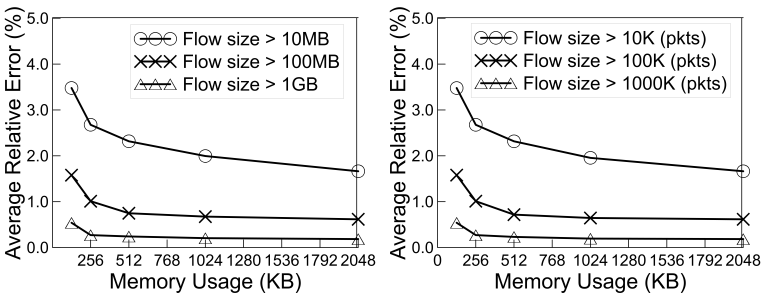


Figure 12: Relative error of giant flow in every second. The number of concurrent flows and packets within the word of flow are shown at the upper of the figure. The giant flow has more than 10^7 packets.

We also recorded the number of concurrent flows and concurrent packets within the same confinement word over time to investigate the accuracy of our noise elimination strategy. As shown in Fig. 12, the relative error of giant flows is stably suppressed around 0.1%,the which means our estimation is robust, regardless of the noise fluctuation over time.

2.6.5 Overheads

Memory. We used the one-hour CAIDA dataset and ran a single-core InstaMeasure to evaluate the estimation accuracy (packets and bytes) while varying the memory usage of FlowRegulator (*i.e.* 128KB-2048KB). Then, we compared each estimated flow size (both in packets and in bytes) with the ground-truth. Since InstaMeasure can measure a flow larger than a million packets, we divided flows into three intervals depending on the size and



(a) Byte counter (b) Packet counter

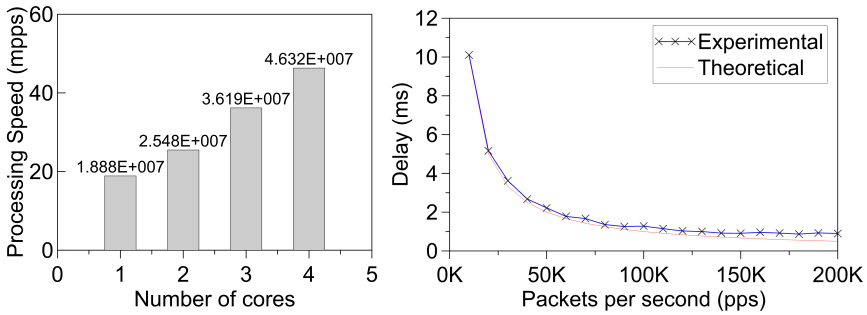
Figure 13: Accuracy of packet and byte counting (CAIDA one-hour trace). Average relative error (ARE) varying memory usage

evaluated the average error of each interval.

Fig. 13(b) shows the average error rates of all L4 flows of the packet counter after a one-hour measurement. When the total memory usage was 128KB, the average error rate of flows that have more than 1000K packets was 0.56% and 1.54% for 100K+ flows. For relatively small flows (10K+ flows), it was 3.48%. As shown in the figure, it decreased as more memory was used. When we increased the memory to 256KB, InstaMeasure achieved 0.28% of average error rate for 1000K+ packet flows, 0.99% for 100K+ flows and 2.79% for 10K+ packet flows. Further, when the amount of memory was 2048KB, InstaMeasure achieved the highest accuracy, with 0.19% (1000K+), 0.58% (100K+) and 1.76% (10K+) error rates, respectively.

Fig. 13(a) shows the average error rates of all L4 flows of the byte counter. When the memory usage was 128KB, the average error rate of 1GB+ sized flows was 0.54%, 1.57% for 100MB+ sized flows, and 3.47% for 10MB+ sized flows. Same as with the packet counter, the accuracy of the byte counter also increased when more memory was given. For 128KB memory, the average error rates were 0.27%, 1.00%, and 2.67%, respectively. For 2048KB of memory, InstaMeasure achieved 0.18% error rate for 1GB+ sized flows, 0.61% for 100MB+ sized flows and 1.66% for 10MB+ sized flows.

Processing Speed of InstaMeasure. To evaluate the encoding speed of InstaMeasure, we used our off-the-shelf device in Fig. 3; it is equipped with an 8-core 2.4 GHz Atom processor and 16G DRAM. We pre-loaded the CAIDA dataset into memory and focused on how many packets InstaMeasure can process per second. Fig. 14(a) shows the processing speed of InstaMeasure by varying the number of cores. As shown, InstaMeasure could process 18.88 mpps (on average) with a single core. Clearly, a one-core InstaMeasure can measure the OC-192 link of the CAIDA dataset even when the traffic is 64-byte packets. The processing speed with two cores increased to 25.48 mpps. Three and four core InstaMeasure still



(a) Speed of InstaMeasure (b) Heavy hitter detection latency

Figure 14: InstaMeasure's processing speed scales well, and its detection latency of heavy hitters is under 1 ms if a heavy hitter consumes more than 100 kpps.

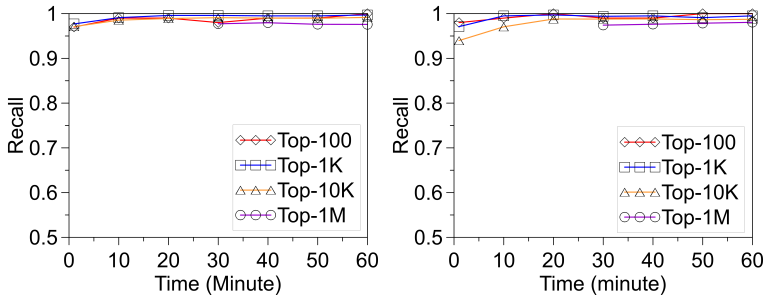
achieved higher processing speed: 36.19 mpps and 46.32 mpps, respectively. We note that InstaMeasure's memory usage does not affect processing speed but only on the accuracy.

In conclusion, this experiment shows that InstaMeasure—even using an Atom processor and DRAM–has enough processing speed that can be sufficiently used for 10 Gbps high-speed links without any packet loss.

Detection Latency. We conducted an experiment to show the heavy hitter detection delay caused by our InstaMeasure's saturation-based decoding in a 1 Gbps network environment. We used a high-end desktop to generate traffic with various speeds (10-200 kpps) to InstaMeasure device. At the same time, our device performed heavy hitter detection in parallel. A fixed threshold (T=0.05% of link capacity) was used to detect heavy hitters and recorded the first detected time using both packet-arrival-based and saturation-based decoding. As shown in Fig. 14(b), when the traffic generator was at a low transmission rate, the detection delay was more than 10 ms. However, as the transmission rate increased, the detection delay decreased sufficiently. When the speed was 10 kpps, the average delay was around 10 ms and 1 ms at the rate of 130 kpps. Note that byte volume-based heavy hitter detection delay is almost the same as with the packet counting-based one. This is mainly because our byte volume counting depends on the packet counting.

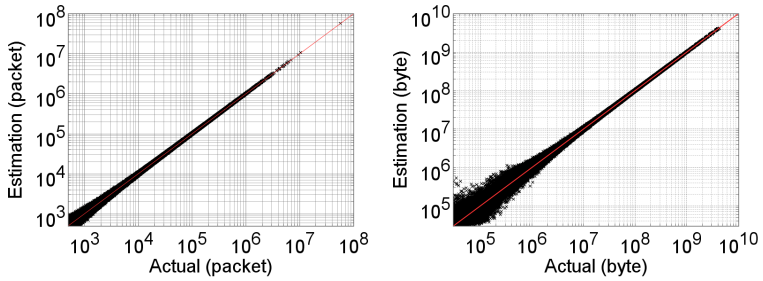
2.6.6 Top-k Identification

Owing to InstaMeasure's high accuracy for millions of flows, the top-k identification problem can be scaled up to Top-million. Moreover, InstaMeasure can provide two kinds of top-k flow lists at the same time: Packet top-k and Byte top-k. For evaluation, we fixed the memory usage of the counter to 10MB and used a standard recall metric to measure the quality of packet number-based and byte volume-based Top-100, 1K, 10K, and 1M lists using the



(a) Packet-based top-k (b) Byte-based top-k

Figure 15: Quality of packet and byte top-k list (CAIDA one-hour trace)



(a) (b)

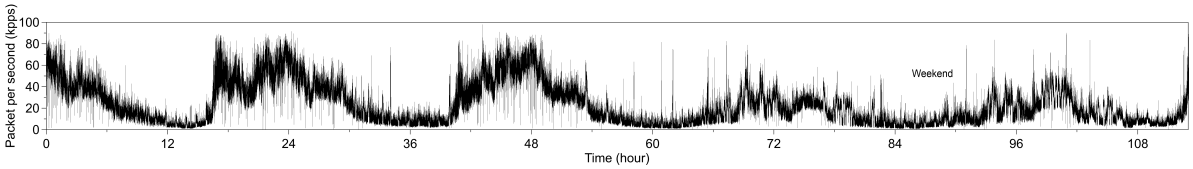
Figure 16: Estimation result of 133 hour real-world experiment using 12MB sketch. Accuracy of packet counting (left) and byte counting (right). Each point stands for each flow. To see how accurate estimation is, check how close every point is to the reference line $y = x$.

CAIDA dataset, with updates are done every 10 minutes. Fig. 15(a) and Fig. 15(b) show that the recall rates of byte/packet top-k are mostly above 95%.

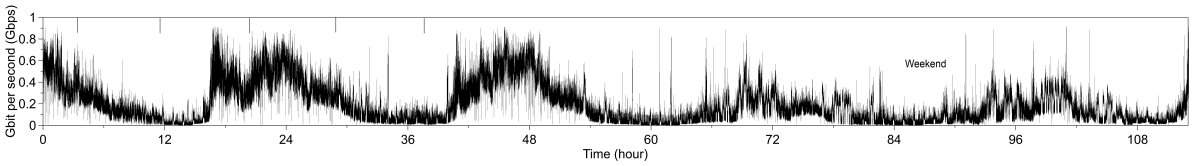
2.6.7 Monitoring in the Wild

We observed that the traffic collected on our campus for 113 hours had the typical Zipf-like distribution as other network traces did. During 113 hours, 9.1 billion packets of 122.3 billion L4 flows were measured simultaneously both in packets and in bytes. InstaMeasure used a single Atom processor core, 128KB for the sketch, and 33MB for the WSAF table. Sketches and WSAF tables are all in DRAM.

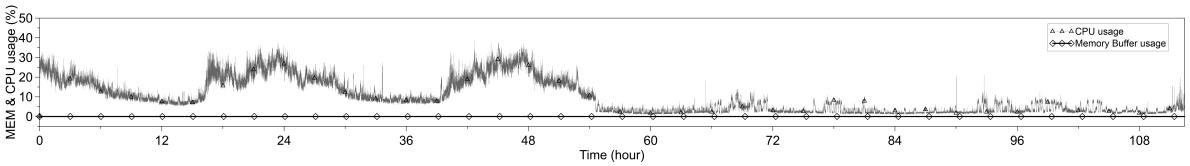
Accuracy. Fig. 29 shows the estimation accuracy by standard error for the real-world experiment. For packet counting, we report 0.54% standard error over 350 flows of which size is 1000K+, 1.61% over 11,047 flows for 100K packets, 3.46% over 104292 flows for 10K+ packets. For byte counting, we report 0.63% over 414 flows of which byte size is 1G+, 1.74%



(a) Packet per second (kpps)

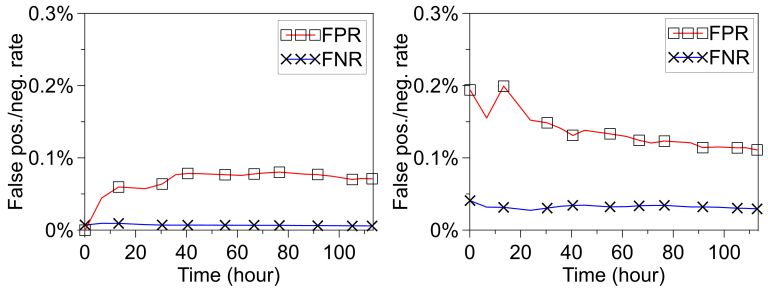


(b) Gigabit per second (Gbps)



(c) CPU load of 1 core InstaMeasure

Figure 17: Monitoring in the wild: Our campus uses 2 Gbps bandwidth in total (1 Gbps for up-link and 1 Gbps for downlink), and the backbone gateway router uses a Juniper EX9208 switch, as shown in Fig. 4.



(a) (b)

Figure 18: False positive and false negative rates of packet heavy hitter detection (left) and byte volume heavy hitter detection (right).

over 12,125 flows of 100MB+, 3.65% over 107,726 flows of 10MB+. This accuracy matches the accuracy observed in the lab experiment with the CAIDA dataset.

Overheads. Our campus network's traffic volumes are shown as a time series in Fig. 17(a) and Fig. 17(b). We observed that the amount of traffic reached a peak during the daytime, whereas less traffic was observed at the weekend and night. InstaMeasure's CPU workload and the queue memory usage during the 113 hours are shown in Fig. 28. the core's workload matches the traffic pattern, and the core usage did not go over 40% at any point. As for the queue (represented in black diamonds in the figure), it did not grow noticeably. The results confirmed that InstaMeasure implemented on the Atom board worked well for the 1 Gbps network monitoring and for quite a long time.

Heavy Hitter Detection. Fig. 18 shows InstaMeasure's heavy hitter detection accuracy in terms of false positive/negative rate. Owing to InstaMeasure capability of counting both in packets and in bytes, it can detect both packet heavy hitters and byte heavy hitters. False-negative rates in both cases are negligible, and the false-positive rates of packet/byte heavy hitters are less than 0.1% and 0.2%, respectively.

2.7 Conclusion

In this work, we have developed InstaMeasure for instant flow detection by counting packets and bytes in high-speed networks. Our approach is different from conventional measurement frameworks in that we reduced detection delay by introducing a new notion of a large In-DRAM working set of active flows (WSAF) table. To deal with the slow access speed of DRAM, we designed a multi-layer sketch-based FlowRegulator to retain flows in front of WSAF for relaxing the influx rate of DRAM. FlowRegulator's design is inspired by a state-of-the-art algorithm, in which we report a technical flaw and provide a better estimation formula to improve its accuracy. Moreover, we extend the design with a multi-layer approach to scale up the counting capacity. Based on our FlowRegulator, we can perform an instant network traffic measurement using large DRAM and can obtain measurement results with under 1 ms detection delay, which is negligibly small compared to tens or even hundreds of milliseconds in the conventional approaches. Further, we built a multi-core instant flow-level measurement system named InstaMeasure and prototype it in an Atom-based off-the-shelf device. Last but not least, we demonstrated the performance and feasibility of our system in both a laboratory setting (one-hour CAIDA trace) and a real-world setting (113-hour campus gateway).

3 RFlow$^+$: An SDN-based WLAN Flow-level Monitoring and Management Framework

3.1 Introduction

With the plethora of WLAN deployments in residential, enterprise, and public settings, the Internet has become more accessible than ever. This proliferation has become even more expedited because of the increasing number of WLAN devices and demands from a wide range of user devices, such as smartphones, tablet PCs, and IoT devices. In order to not lag behind users' aggressive network bandwidth demands in their daily lives (*e.g.* for YouTube or Netflix), WLAN technologies have rapidly advanced in terms of bandwidth: 802.11n [37] (up to 600 Mbps), 802.11ac (up to 6.933 Gbps), and so on [89]. Moreover, the Multi-user Multiple-Input and Multiple-Output (MU-MIMO) feature was added into the IEEE 802.11ac wave 2 certification to increase WLAN's multi-user support capacity [33]. Interestingly but unfortunately, despite the advancements of WLAN technologies, people are easily dissatisfied with their WLAN infrastructures due to a bandwidth throttling from WLAN service providers [30].

The reasons for this dissatisfaction are two-fold: (1) an absence of intelligent and timely network management followed by (2) the limited view of network traffic monitoring tools (*e.g.* NetFlow [12] and sFlow [85]) and vendor-oriented configurability. Instead of näive over-provisioning of access points (APs), we can provide users with more stable and thus more reliable network conditions (*e.g.* latency, jitter, and required minimum bandwidth) by accurate network monitoring and timely treatments such as rate-limiting, the access control list (ACL), or flow quarantines.

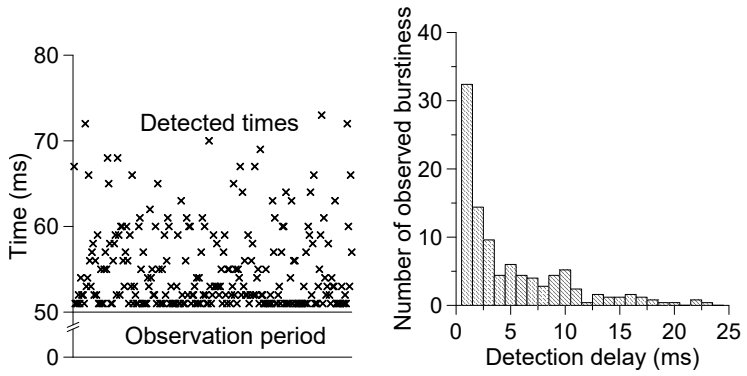
Even though network traffic monitoring and management solutions are dominated by major vendors, those vendors focus mainly on the core switch, and rely on either advanced hardware (*e.g.* TCAM and SRAM in NetFlow [12]) or sampling approaches (sFlow [85]). For an SDN enabled core switch, OpenFlow is an additional option to perform monitoring tasks. Note that any monitoring solution falls in one of these three models. Recently, intense efforts in two main streams have been made to realize the "*victory*" of SDN-driven data centers like B4 [39] in the WAN domain. First, efforts have been made in WLAN management frameworks. Unlike OpenFlow [60], a *de facto* standard interface between a controller and switches, a WLAN management framework requires additional features such as wireless channel selection, interference mitigation, and mobility management. To achieve these, BeHop [95], Odin [82], and OpenSDWN [81] customized OpenFlow's configurability for WLAN by introducing the concept of virtual APs. The other optimization efforts have addressed

WLAN monitoring frameworks (*e.g.* PayLess [11], OpenSketch [98], FlowSense [97], and OpenTM [90]). These monitoring frameworks tried to overcome the intrinsic limitations (*i.e.* the limited accuracy of default settings and resource-hungry nature of full sampling) of generic sampling-based solutions.

In this paper, we aim to design an SDN-based flow-level monitoring and management framework for WLAN. In terms of monitoring, and unlike a general network monitoring framework, WLAN additionally requires the framework to monitor wireless network traffic at different target levels (*i.e. short-term* bursty users and *long-term* heavy down-loaders/up-loaders) because of its openness and users' dynamics. Moreover, the price of the WLAN device is much lower than the core switches. Thus, the expensive hardware (*e.g.* TCAM and powerful CPU) are unlikely to be embedded. These constraints result in the computational heavy NetFlow-like solutions are impractical for a WLAN device. Also, in terms of management, the framework needs to improve the overall wireless bandwidth utilization by the flow-level timely resource allocation actions (*i.e. immediate* action according to short-term monitoring results and *eventual* action according to long-term monitoring) as well as accommodate more users by dynamically providing capacity. To our best knowledge, no existing studies have ever included both approaches in their design considerations.

To cope with these issues, we propose RFlow$^+$ to achieve two different levels of network monitoring—local (switch/AP level) and global (controller/collector level); thereby supporting application-specific actions (*i.e. immediate* and *eventual*) via a network management framework. The recyclable counter with confinement (RCC) [66] motivates RFlow$^+$'s major design as it provides reliable counting accuracy while efficiently managing its memory usage; this consequently reduces network overheads, as further detailed in Section 3.2. Mainly because of our two-level (*i.e.* global and local) monitoring framework design based on the RCC counter, RFlow$^+$'s major departure from existing work is that the *local agent* takes the first step toward supporting immediate actions (*e.g.* flow rate-limiting or flow quarantines), which can be flexibly managed by users/operators' high-level descriptions (see Section 3.4.2). We consider our solution to be an addition to the existing WLAN management frameworks, but not an alternative. Also, our framework is totally independent of any other management tasks, and can run as a module of any existing solutions.

Contributions. In this paper, we are making the following contributions: First, we propose RFlow$^+$, a novel monitoring and management framework for WLAN, to support both *short-term* and *long-term* monitoring applications and enforce timely treatments (*i.e.* rate-limiting and flow quarantines) based on their requirements (*i.e. immediate* and *eventual*). The counting algorithm performs short-term measurement (*e.g.* 50 ms time window) locally as well as long-term measurement (*e.g.* one month) globally. Second, we propose an online decodable sketch-based multi-tenant cardinally measurement algorithm for a resource-



(a) Distribution of detected times

(b) Distribution of delay

Figure 19: Distribution of burstiness detections with RFlow$^+$ (250 trials)

constrained device (*e.g.* WLAN device) to overcome the threat of the flow table overflow attack in an open setting. Third, we propose a pipeline design of sketches and prototyped RFlow$^+$ on top of OpenWrt on off-the-shelf access point hardware (TP-Link AC1750) as add-ons on OpenVSwitch (OvS) [73] and OpenDaylight [67]. Last, we evaluated RFlow$^+$ by conducting extensive experiments, such as short-term/long-term per-flow measurement, flow table overflow detection, detection delay evaluation, and CPU overhead, among other metrics. To show the feasibility of RFlow$^+$, we performed real-world experiments by deploying our APs on our university campus.

3.2 Motivation

In this section, we discuss why in-network monitoring tools need to support two different types of flow measurements—local (switch/AP level) and global (controller/collector level)—based on the timeliness. Consequently, we introduce a novel monitoring and management framework called RFlow$^+$ that fits such requirements and further supports timely treatments (*i.e.* executing predefined immediate action rules). There are two different monitoring applications, namely long-term (*e.g.* heavy down/up-loaders and ISP billing) and short-term (*e.g.* bursty traffic [52, 80] and MAC flooding). Last, we discuss the potential threats of the SDN-enabled WLAN device in a wild environment.

3.2.1 Long-term Monitoring

To rate-limit heavy down/up-loaders or for ISP billing, we might need to measure in-network flows for a long period (say, a week or month). For accuracy and consistency, these measurements should be performed in a global manner. That is, aggregate statistics from every AP should be used for the final decision. NetFlow and sFlow have been widely adopted to monitor wired and wireless in-networks. However, these sampling-based monitoring solutions lead to low accuracy with per-flow counting (even missing transient flows), while increasing the sampling rate requires too many resources (*e.g.* CPU, memory, and network bandwidth).

With its advent, SDN technology provided control of the data plane from the controller by adding/removing forwarding rules in the range of Layers 2–4 and further provided resource-efficient statistics at different aggregation levels (*e.g.* `flow`, `port`, and `table`). These statistics are collected from OpenFlow enabled switches or OvS via the OpenFlow (hereafter, native-OF). The native-OF not only counts packets and bytes per-flow, but it also provides an aggregated view of the statistics of `table` and `port`. In reality, however, the higher the layer at which the flow is defined, the more flows are generated. When the SDN controller periodically sends an `OF_FLOW_STAT_REQUEST` message to the OpenFlow switch/OvS, the `OF_FLOW_STAT_REPLY` message that contains entire flow entries of userspace flow tables should be sent back to the controller, which is the first weak point of native-OF: *lack of scalability*. Second, native-OF's flows are defined mainly for routing purpose, which means the flows are defined in a highly aggregated manner (*e.g.* in layer 2 or 3). However, an application may require statistics on some specific layer-4 flows while flows of interest are not determined before the request. To reply to the request, the flows should previously be defined with all possible combinations of Layers 2–4 addresses due to the fact the native OF provides statistics only for the defined flow. These limitations cause significant CPU[1]/network overheads (See Section 3.6.1).

To monitor flows efficiently and in a timely manner (one of the RFlow$^+$'s goals), we adopt RCC, an approximate counter. Nyang *et al.* introduced RCC in [66] as a low-memory-cost approximate packet counter designed for large-scale and real-time per-flow measurement in a high-speed router. RCC achieved its high accuracy (approximately 99%) using a small-but-recyclable virtual vector and obtained high-speed access by confining virtual vectors within a CPU word instead of spreading them over the entire memory. Additionally, RCC provides two desirable features: (1) RCC decoding can be performed in real-time (about two hash computations per decoding operation), and (2) RCC provides the top-K list. RCC inspired RFlow$^+$, which resolves the two formerly mentioned issues of native-OF: lack of scalability and non-generic statistics RFlow$^+$ allows the OvS to define the minimum number of flows;

[1]Giotis *et al.* proved that CPU overhead is also caused by high-level flow definition [28]; thus defining a larger number of flows at high levels finally leads to additional overheads in both CPU and network resources.

Thereby minimizing network overheads caused by statistical reports. Additionally, it provides generic statistics on every possible flow without requiring a long list of flow definitions, thanks to RCC. As RCC can track the estimated packet counts with high precision, RFlow$^+$ at a switch reports only non-zero entries in *elephant-flow* counter to a central collector, as those flows are active (*i.e.* regions of interest) for a statistical collection period (*e.g.* a default setting of three seconds in native-OF). By doing so, RFlow$^+$ can reduce network overheads as well as achieve memory efficiency. Of course, locally-made microscopic statistics (*i.e. mice-flow* counters) can be sent to a central collector on-demand or periodically. Note that the size of overall statistical message exchanges over the network is significantly small (see Section 4.5). Finally, RFlow$^+$ performs *eventual* (not *immediate*) actions (*e.g.* limiting or blocking monthly heavy down/up-loaders, advanced persistent threat attackers, or slow scanning attackers and ISP billing) based on the collected long-term statistics.

3.2.2 Short-term Monitoring

As well as long-lived flows, in-network overall flows contain short-lived flows. Unlike long-lived flows, short-lived ones are transient. Thus, the flow measurements and their treatments should take place on-site and in real-time. Short-term bursty traffic causes other users in the same network to experience degraded network performance or intermittent disconnections. Sarvotham *et al.* reported that bursts are not caused by a "conspiracy" of many moderate flows, but rather by a few dominating connections (*i.e.* alpha traffic [80]). Thus, it is beneficial to immediately limit the dominating connections/flows to avoid saturating the link. As this type of activity is very short-term, it is hard (or impossible) to detect it with existing monitoring frameworks. Although detected, its treatment can be *post-mortem*. To remedy this, the primary goals of RFlow$^+$ are (1) to design a network monitoring system that can detect these transient events, and (2) to provide a local flow regulation in an instantaneous manner. Here, the "+" in RFlow$^+$ means that RFlow$^+$ can apply an immediate action on a switch in a pre-defined manner—for rate-limiting, flow quarantines, and other actions.

To correctly identify the dominating flows and limit them accordingly, reliable burstiness detection must first be implemented. To our best knowledge, none of the existing monitoring solutions can detect burstiness in real-time. Lan *et al.* proposed three definitions of burstiness in an offline measurement study [52], namely variance, round trip time (RTT), and train burstiness. As RTT burstiness is not detectable because of the existence of unidirectional flows, RFlow$^+$ adopts the variance definition of burstiness, as our burstiness decision should be made on-the-fly and its reported accuracy is qualitatively similar to train burstiness. Variation burstiness is based on the variation of traffic at a time-scale of T[2]. Given a flow,

[2]In our settings, we set $T = 50$ ms as in [80] and ignore flows shorter than T, as their variance is undefined.

it is divided into bins b_i and the number of bytes sent in b_i is defined as s_i. The variance burstiness of the flow is then defined as the standard deviation of all s_i. Fig. 19 shows the distribution of burstiness detections obtained by RFlow$^+$. Among 250 trials with $T = 50$ ms, RFlow$^+$ provides a 100% detection ratio within 23 ms delay. Table 2 presents a summary of how RFlow+ provides a good trade-off between accuracy and cost savings and supports both short-term and long-term applications. As mentioned earlier, none of the existing monitoring tools can detect transient network anomalies nor regulate them in real-time.

3.2.3 Deployment In the Wild

One of the goals of SDN is to leverage a global view and centralized control for realizing a dynamic resource allocation and network control. The concept of reactive flow plays an essential role, where OpenFlow allows a controller to install flow rules, either proactively or reactively. Unlike the proactive approach which pre-installs flow rules before traffic's arrival, the reactive approach supports a dynamic forwarding behavior of switches by querying forwarding decision from the controller. The controller uses either `PacketOut` to provide a one-time decision or install a flow rule to switch using `FlowMod` message. Later, the installed flow is automatically removed based on two timeouts: *hard timeout* or *idle timeout*.

Even though the SDN is expected to overcome legacy network problems (*e.g.* switch loop, failure, and congestion etc.); however, SDN itself is facing a serious threat, namely the resource saturation attacks: bandwidth hogging and flow table overflow. In the past few years, researchers have reported various similar attacks, either at a high rate [19, 47, 74] or low rate [72, 86]. The main goal of these attacks is to dysfunction a switch or affect benign traffic by exhausting the switch's resource using flow installation event (*i.e.* flow table overflow attacks). Since the reactive application in a controller is sensitive to 5-tuple [101], adversaries are assumed to trigger flow installation events as much as possible by sending packets with randomly generated 5-tuples.

Several solutions have been suggested to defend against flow table overflow attack [19, 26, 75, 99, 100]. However, these solutions are designed for backbone switches. Unlike the powerful core switches, WLAN switches are much cheaper; thus, having limited resources to support the computationally-heavy measurement tasks. In the last decade, sketches (compact data structures) are suggested to perform measurement tasks in a resource-constrained environment [17, 93, 96]. Due to their designs, the encoding process is highly efficient, whereas the decoding process is computational heavy. For this reason, a powerful remote decoding server is commonly used in the sketch-based systems [55, 57, 98], which leads to detection, and the corresponding treatments are delayed due to a control loop. As we discussed in subsec-

To avoid errors from boundary effects, we also ignore flows shorter than 3–5 T.

Table 2: Comparison of RFlow$^+$ with Native-OF and sFlow regarding performance of Local Short-term Monitoring (LSM) and Global Long-term Monitoring (GLM). Metrics include memory, CPU, measurement accuracy, detection responsiveness, scalability and implementation cost are used to show the trade-off between these three approaches.

	Counting type	Native-OF (Exact)	RFlow$^+$ (Approximate)	sFlow (Sampling)
LSM	Memory	Proportional to the number of defined flows	2 MB for more than 100K flows (including for long-term detection)	N/A
	CPU	Proportional to the number of defined flows	Proportional to the number of active flows $\geq$ 1 hash ($\approx$ 30 *clocks*) for per-flow detection	N/A
	Accuracy	Exact	Standard error$\leq$ 5%	N/A
	Responsiveness	Not supported	Real-time	Not supported
	Implementation	Needs data structure redesign	Provided (200 lines of code)	Needs system redesign
GLM	Accuracy	Exact	Standard error $\leq$ 1%	6% or higher sampling rate for standard error $\leq$ 2%
	Scalability	Proportional to the number of defined flows	Proportional to the number of active flows over a collection period	According to the polling interval and the sampling rate
	Implementation	Needs additional memory	Provided (200 lines of code)	Provided

tion 4.1, the mobility and dynamics of the WLAN environment highly require the WLAN management framework to perform the detection and control in a timely manner. To realize a real-time detection of flow table overflow attack, we propose a sketch-based online decodable multi-tenant cardinality measurement algorithm to overcome the threat of WLAN SDN device from the flow table overflow attack in the wild. We combined our sketches in a pipeline design and deployed our framework into a real WLAN device. Through extensive experiments, we found our sketches are highly accurate and feasible in a resource-constrained device (See subsection 4.5).

3.3 Related Work

RFlow$^+$ falls into two fields of SDN-based frameworks, namely management and monitoring.

3.3.1 SDN-based WLAN Management Frameworks

In broadband access networks, bandwidth allocation of flows is an important problem, as it degrades the overall network performance. One reliable solution is bandwidth allocation based on the application type. To achieve this, Seddiki *et al.* proposed FlowQoS, which contains two modules, namely a flow classifier and an SDN-based rate limiter [84]. FlowQoS, implemented on top of OpenWrt, demonstrates enhanced performance for both adaptive video streaming and VoIP in home settings in the presence of active competing traffic. However, this work has a limited view of the in-network traffic monitoring compared to RFlow$^+$, and its offloading technique for traffic classification on the controller is inherently limited to short-term monitoring applications.

3.3.2 SDN-based WLAN Monitoring Frameworks

Because of their generic support for different measurement tasks, NetFlow and sFlow have been widely adopted to monitor wired and wireless in-networks. However, these sampling-based monitoring tools cannot accurately report counts per flow. To better cope with this, a fair number of network monitoring tools based on OpenFlow have been proposed recently. OpenTM uses OpenFlow's built-in per-flow statistics reported from OpenFlow switches to directly and accurately measure the traffic matrix with low overhead. OpenTM exploits the routing information obtained from the controller to choose flow statistics of interest intelligently, thereby reducing the load on switching elements [90]. Yu *et al.* proposed FlowSense [97], a push-based monitoring tool that exploits passive (not on-demand) update messages sent by OpenFlow switches to the controller to inform it of in-network changes

(*e.g.* `PacketIn` or `FlowRemoved` messages) and to efficiently infer link utilization in flow-based networks. However, these solutions are limited to transient traffic behaviors (*i.e.* burst traffic), as they did not consider short-term monitoring applications.

To support long-term monitoring applications with eventual actions, Yu *et al.* also proposed OpenSketch, which separates the measurement data plane from the control plane [98]. In the data plane, OpenSketch provides a simple three-stage pipeline (hashing, filtering, and counting) to support many measurement tasks (*e.g.* heavy hitters [17], DDoS, flow size distribution [49], traffic change detection [83], and count traffic). In the control plane, OpenSketch provides a measurement library that automatically configures the pipeline and allocates resources for different measurement tasks. Chowdhury *et al.* proposed PayLess [11], a monitoring framework for flow statistics collection at different aggregation levels. To further enhance effectiveness, PayLess uses an adaptive scheduling algorithm (*i.e.* variable rate sampling based on link utilization) for flow statistics collection.

However, RFlow$^+$'s major departure from existing solutions is the practical consideration for WLAN monitoring and management from its design space; thereby supporting both *short-term* and *long-term* flow-level monitoring applications and enforcing treatments (*i.e.* rate-limiting and flow quarantine) based on their requirements (*i.e.* immediate or eventual).

3.4 RFlow$^+$ framework Design

In this section, we demonstrate the high-level design of RFlow$^+$. First, we describe our framework at a high level. Second, we introduce the components of our RFlow$^+$ local agent and RFlow$^+$ global agent individually. Last, we present the algorithm of our local agent functions.

3.4.1 Overview

As shown in Fig. 20, RFlow$^+$ extends a general SDN framework with a RFlow$^+$ global agent in the collecting layer and RFlow$^+$ local agent in the infrastructure layer. The application layer interacts with the SDN controller in the control layer and RFlow$^+$ global agent in the collecting layer to provide flow management and monitoring, respectively, for user billing, security functions, heavy user detection, and quality of service (QoS). The SDN controller also retrieves statistics collected from the OvS via the northbound API. OpenDaylight, a popular SDN controller, resides in the control layer. It provides flow management and statistics collection APIs for northbound applications. Through the southbound API, the SDN controller sends control/data plane messages and requests for statistics to the OvS. The RFlow$^+$ global agent is located in the collecting layer. It stores the statistics received

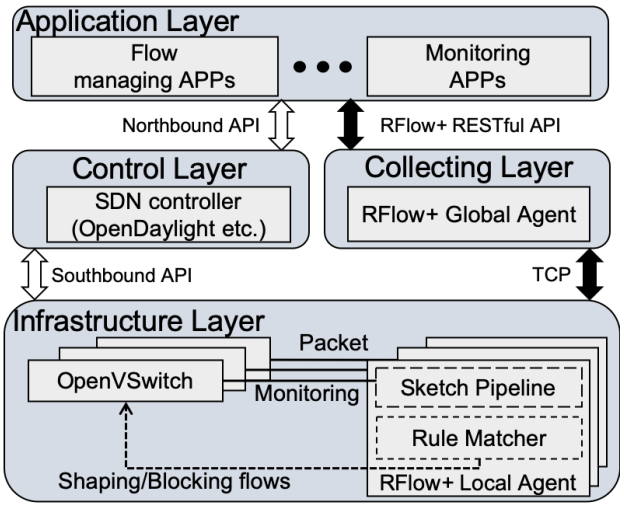


Figure 20: Architecture: RFlow$^+$ extends a general SDN framework with a RFlow$^+$ global agent in the collecting layer and RFlow$^+$ Local Agent in the infrastructure layer

from the RFlow$^+$ local agents periodically, and it also obtains overall statistics (including macroscopic statistics) on demand. It provides a RESTful API for northbound applications to access statistics or to propagate predefined immediate action rules to the OvS. Finally, RFlow$^+$ local agent is located in the infrastructure layer. It is responsible for per-flow packet counting and executing predefined immediate action rules populated by the application layer. Note that flows in RFlow$^+$ are not the same flows defined in native-OF, but flows defined in Layers 2—-4.

3.4.2 RFlow$^+$ Local Agent

Figure 21 shows the internal working flow of RFlow$^+$'s local agent. RFlow$^+$'s local agent consists of four components: a sketch pipeline, a local flow record table, a predefined rule table, and a rule matcher. As shown in Figure 21, the sketch pipeline is associated with an OvS for monitoring packets that go through it. Sketches continuously update the flow records that are temporarily stored in a local flow record table. Finally, the rule matcher maintains a predefined rule table and regulates flows using the statistics provided by the local flow record table.

Sketch Pipeline. *Sketch Pipeline* is composed of several compact data structures (*i.e.* sketch). Here, sketches are used as building blocks to meet a diverse demand in measurements (*e.g.* L2/L3/L4 per-flow measurement, heavy hitter, super-spreader detection. DDoS attack detection, etc.). In this paper, we use two different sketches, namely the recyclable counter

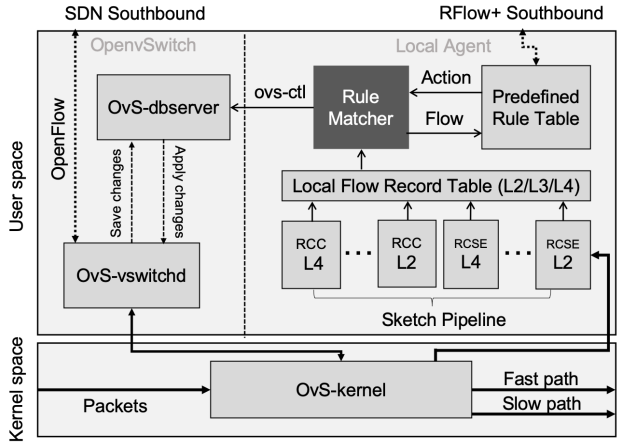


Figure 21: Internals of RFlow$^+$ local agent. Sketches monitor OvS kernel traffic and continuously update the flow records to the local flow record table. The rule matcher maintains a predefined rule table and regulates the flows combine with flow records.

with confinement (RCC) [66] and the compact spread estimator (CSE) [96]. Both of those techniques are based on linear counting (LC) [93].

RCC was introduced by Nyang *et al.* for performing per-flow traffic counting with small CPU/memory overheads in a high-speed link. The online encoding/decoding capability and the accuracy of RCC benefit our framework in performing online measurement for launching immediate actions based on predefined rules. In RFlow$^+$, simultaneous traffic monitoring of layers 2–4 is supported by constructing three RCC sketches in a pipeline called RCC-L2, RCC-L3, and RCC-L4. These sketches use different fields from the packet header because inputs depend on the layer of the measurement task. For instance, RCC-L2 uses MAC address pairs (*i.e.* source, destination), RCC-L3 uses IP address pairs, and RCC-L4 additionally requires protocol and port pairs as inputs.

While RCC provides high efficiency and accuracy in per-flow measurements, it cannot provide the spread size of the source or destination. Since spread measurement is also crucial for defending against a flow table overflow attack, we employ CSE to compensate for RCC's inability to spread estimation. CSE is designed for counting distinct elements in multi-tenant (or multi-set cardinalities), where the original LC is for counting distinct elements in a single set cardinality. Based on the theory of LC, CSE showed high accuracy in the multi-tenant cardinality measurement. However, it requires a lot of memory access for decoding, which becomes a bottleneck in online detection.

Revised CSE. To address the aforementioned problem, we introduce a revised CSE (RCSE) that combines the ideas of RCC and CSE to perform multi-tenant cardinality measurement

to detect a flow table overflow attack. The key idea of our RCSE is to confine virtual vectors in a consecutive memory space of which the size is equivalent to the size of a CPU's cache line (*e.g.* 32 bytes or 64 bytes) for reducing the overhead due to mass memory accesses. For example, the worst case of the original CSE when using a 32-bit virtual vector is reading 32 words for decoding (*i.e.* read 1 bit per word). However, if we confine the virtual vector in a small consecutive memory space (*e.g.* four words), we only need up to 4-word readings to complete the task. As most sketches, RCSE also is subject to the virtual vector saturation problem, meaning that the virtual vector reaches the maximum capacity. In RCSE, we set the maximum counting capacity of the virtual vector as the threshold of the attack detection. Once a virtual vector triggers the saturation event, we reset the virtual vector to 0 for the next round detection. As shown in Section 3.6.4, RCSE provides a good trade-off between accuracy and memory usage, and the threshold of detection is adjustable by changing the size of the virtual vector. When constructing the sketch pipeline, we use three RCSE sketches (*i.e.* RCSE-L2, RCSE-L3, and RCSE-L4) for performing layer-2 to layer-4 flow table overflow detections in parallel. RCSE requires the same inputs as RCCs do in each layer.

Overall, *Sketch Pipeline* is associated with the OvS for monitoring packets. Depending on the requirements of measurement tasks in the different layers, different fields (*i.e.* MAC address, IP address, protocol, port number) of a packet header are extracted and passed to the sketches for detecting attacks and updating statistics (*i.e.* flow records) of the local flow record table (Layers 2–4) in parallel. RFlow$^+$ provides an option to enable/disable measurement in each network layer by adding or removing sketches in the sketch pipeline.

Local Flow Record Table. *Local Flow Record Table* is the temporal storage of flow records reported from the sketch pipeline. These flow records are used to launch an instant measurement task for performing immediate actions according to predefined rules. Meanwhile, the local agent periodically updates and accumulates these locally stored flow records to a global agent for performing server-side long-term monitoring. The list of non-zero flow records are encoded in a JSON format and are sent to the collecting layer through transmission control protocol (TCP). After sending, the local counter table resets the packet counts of the entries to zero in the local flow record table, as if we updated only the active flows' statistics. A `NodeID` is assigned to each of our local agents by the SDN controller, and is sent along with the list of the flow records to distinguish the updates from different devices. Up to three flow record tables were dynamically allocated to meet measurement tasks in different layers (*i.e.* Table-L2, Table-L3, and Table-L4). It is worth mentioning that *Local Flow Record Table* is a general hash table. Since our sketch (*i.e.* RCC) can efficiently reduce the burden of the hash table [42], RFlow$^+$'s traffic measurement tasks fulfill the online processing requirement, as shown in Fig. 25.

Predefined Rule Table. *Predefined Rule Table* is a storage for the immediate action rules

that predefined by northbound applications. The entry of the rule table consists of two parts: the matching field and the action field. The matching field is used for matching flows queried by the *Rule Matcher*, and the action field indicates the corresponding immediate actions should be applied for queried flows. In the background, a *Rule Manager* is responsible for receiving and installing the immediate action rules in the predefined rule table. **Rule Matcher.** *Rule Matcher* is mainly responsible for executing the predefined immediate action rules according to the statistics provided by the local flow record table. The *Rule Matcher* continuously tracks the update event of each flow record and checks if flows match one of the fields in the table. Northbound applications predefine the immediate action rules and populate them via APIs provided in RFlow$^+$'s global agent. There are various ways to limit the rate of flow: deleting the flow, associating a flow with a QoS queue, setting flow actions as "drop" or redirecting the flow to bandwidth-limited paths. Of course, we can also limit sending/receiving rates of an interface via other network traffic control tools (*e.g.* Queuing Disciplines (qdisc) [71]) that are supported by OpenWrt [68] (the OVS runs on top of it).

3.4.3 RFlow$^+$ global agent

The RFlow$^+$ global agent resides in the collecting layer to store the statistics obtained from local agents, as shown in Fig. 22. Northbound applications can further collect statistics from the RFlow$^+$ global agent and can propagate immediate action rules (*i.e.* high-level descriptions) to the OvS through the RFlow$^+$ RESTful API. The RFlow$^+$ global agent consists of the following five modules:

- **Global Flow Record Table:** The global flow record table is persistent storage that has the same structure as the local flow record table in the RFlow$^+$ local agent and is distinguished by switch NodeID.
- **Node Selector:** This supports customized statistics at the switch level. Users can choose for single, multiple, or all nodes to collect statistics, as shown in Fig. 23. Besides, the node selector can interact with the other modules of the global agent to obtain combined statistics.
- **Flow Selector:** This module is responsible for aggregating flow statistics from the global counter table. By giving a partial flow definition, high-level primitives also can be collected (*e.g.* TCP port 2424). The flow selector module can interact with the node selector module to collect statistics from specific switches.
- **Layers 2–4 Aggregator:** This module aggregates statistics among different layers. For example, layer-2 (L2) Aggregator may want to aggregate statistics from the MAC-level, L3 at IP-level, and L4 at Port-level. By selecting different combinations, the

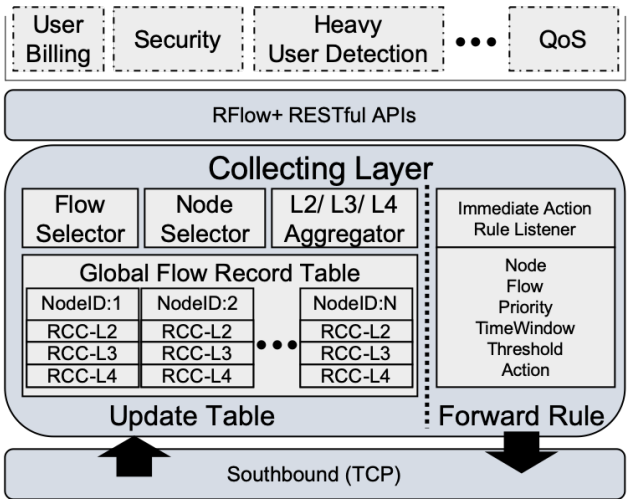


Figure 22: Internals of RFlow$^+$ Global Agent. The global flow record table collects and stores the flow record updates from the local agent for providing statistics accesses to northbound applications through RESTful API.

global agent can produce customized statistics to satisfy different applications' demands.

- **Immediate Action Rule Listener:** This module is responsible for listening for predefined immediate action rules (from northbound applications) via RFlow$^+$ APIs and forwarding the rules to a flow management unit located in the RFlow$^+$ local agent; thereafter enforcing the rules on the switch.

3.4.4 RFlow$^+$ RESTful API

RFlow$^+$ provides a RESTful API for northbound applications to access statistics or populate switches with predefined immediate action rules. Every network application needs to create a JSON object called `RFlow+Request` (see Fig. 23) to customize statistics according to their purpose and define immediate action rules. `RFlow+Request` contains the following:

- **Type:** The network application needs to define what type of operations it wants (*e.g.* retrieval of long-term statistics) and define immediate action rules.
- **Node**: This designates the set of switches to be managed.
- **AggregationLevel:** The network application enables execution on a specific layer defined in the Type field. For example, if Type is "statistics," the entire table entries

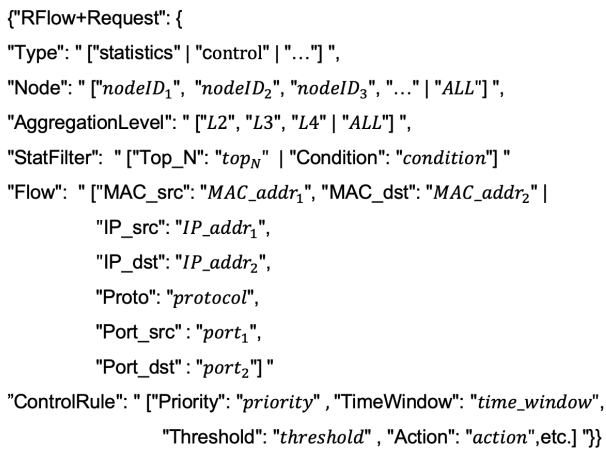


Figure 23: `RFlow`$^+$`Request` object

will be returned according to the selected layer (*e.g.* L2, L3, or L4). If Type is "control," immediate action rules for specified layers will be defined.

- **Flow:** This expresses a specific flow for statistics collection. The seven variables used to express a flow are `MAC_src`, `MAC_dst`, `IP_src`, `IP_dst`, `Proto`, `Port_src`, and `Port_dst`. Each variable is set according to the selected AggregationLevel. For the "statistics" type, an application can specify the partial (or entire) flow definition for accepting high level primitives.
- **StatFilter:** This field is available for a "statistics" type request to filter statistics with parameters (*e.g.* `Top_N` and `Condition`).
- **ControlRule:** This field is available for a "control" type to express an immediate action rule with parameters: `Priority`, `TimeWindow`, `Threshold`, `Action` and other parameters.

3.4.5 Algorithm Design

Once the connection is established between RFlow$^+$ global agent and local agent. The local agent starts the iterative monitoring process with the user-inputted `Interface` to capture its packets. When a packet is captured, it extracts information of different layers: `MAC, IP, Proto, Port` from packet header. The extracted information ($\texttt{Flow}_{\texttt{Info}}$) is passed to *Sketch Pipeline* for approximate counting.

As described in subsection 3.4, our sketch pipeline uses two different sketches (*i.e.* RCC and RCSE) for providing layer-2 to layer-4 measurements, simultaneously. Algorithm 1

Algorithm 2: Sketch Pipeline

```
   input: Interface, RCSE[], RCC[]
 1 forall Pkt_f from Interface do
 2     Flow_Info ← Extract_five_tuple(Pkt_f);
 3     if L4_RCSE_enabled then
 4         VirtualVector ← RCSE_encode_L4(RCSE[], Flow_Info);
 5         if VirtualVector saturation then
 6             Action ← Rule_table_lookup(Flow_Info);
 7             OvS_system_call(Flow_Info, Action);
 8             Recycle(VirtualVector);
 9         end
10     end
11     if L4_RCC_enabled then
12         VirtualVector ← RCC_encode_L4(RCC[], Flow_Info);
13         if VirtualVector saturation then
14             est ← RCC_Decode(VirtualVector);
15             Rule_Matcher(Flow_Info, est);
16             Recycle(VirtualVector);
17         end
18     end
19 end
```

shows an example of how our sketch measures layer-4 traffic. Note algorithms for layer-2 and layer-3 measurement are the same as Algorithm 1, although they require independent memory space and different flow information depending on the layers for which they are utilized. In RFlow$^+$, RCSEs are used for detecting flow table overflow attacks in each layer. Meanwhile, RCCs are responsible for per-flow measurement in different layers. Each sketch is a building block of the pipeline, so each component can be disabled by removing the sketch from the sketch pipeline. *Sketch Pipeline* first extracts $\texttt{Flow}_{\texttt{Info}}$ of different layers, and then it encodes the flow in a compact memory space, which is independently assigned to each sketch. This procedure is executed repeatedly until one of `VirtualVector`'s is saturated. The `VirtualVector` saturation event has different meanings in RCC when compared to RCSE. The saturation event of RCSE means that a source address communicated with more than `est` destination addresses, and thus we have to report the source address to the collector. Also, `VirtualVector` needs a recycling process to perform the next round measurement. Note that the threshold of reporting flow table overflow attacks is adjustable by configuring the size of the virtual vector. For RCC, the saturation means the virtual vector cannot count packets anymore, meaning that it is time to accumulate the estimated

Algorithm 3: Rule Matcher

input: $\texttt{Flow}_{\texttt{info}}, \texttt{est}$

1 $\texttt{FlowRecord} \leftarrow \mathit{Flow\_record\_update}(\texttt{Flow}_{\texttt{Info}}, \texttt{est})$;
2 $\texttt{TimeWindow}, \texttt{Threshold}, \texttt{Action} \leftarrow \mathit{Rule\_table\_lookup}(\texttt{Flow}_{\texttt{Info}})$;
3 /* Short-term heavy user detection */;
4 **if** $\texttt{CurrentTime}() - \texttt{FlowRecord}.Time \geq \texttt{TimeWindow}$ **then**
5 $\quad \texttt{est}_\texttt{T} \leftarrow \dfrac{\texttt{FlowEntry}.est_T \times \texttt{TimeWindow}}{\texttt{CurrentTime}() - \texttt{FlowRecord}.Time}$;
6 $\quad$ /* Flow managing */;
7 $\quad$ **if** $\texttt{est}_\texttt{T} \geq \texttt{Threshold}$ **then**
8 $\quad\quad \mathit{OvS\_system\_call}(\texttt{Flow}_{\texttt{Info}}, \texttt{Action})$;
9 $\quad$ **end**
10 $\quad \texttt{FlowRecord.est}_\texttt{T} = 0$;
11 $\quad \texttt{FlowRecord.Time} = \texttt{CurrentTime}()$;
12 **end**

(decoded) number (*i.e.* `est`) of the saturated `VirtualVector` to the local flow record table by sending $\texttt{Flow}_{\texttt{Info}}$ and `est` to *Rule Matcher*. Then, the `VirtualVector` is recycled for next round counting.

As shown in Algorithm 2. The *Rule Matcher* is responsible for two tasks: flow record update and flow management. *Local Table Update* is used to update the counter in the local flow record tables and returns the flow record entry (`FlowRecord`) (line 1). Also, it is responsible for matching the flow record with the predefined immediate action rule table (line 2). Once a flow triggers one of the rules in the *predefined rule table*, *Rule Matcher* immediately performs flow management using the corresponding action by calling a system call on OvS (line 8).

In the following, we use a case that short-term heavy user detection to explain how the *Rule Matcher* works. `FlowRecord` maintains two counters for different purposes. The first one accumulates the number of packets in a statistic collection period for updating to the global table (*i.e.* `FlowRecord.est`), the other one for measuring the traffic in a time window (*i.e.* $\texttt{FlowRecord.est}_\texttt{T}$. After updating these two counters, `Rule_Matcher` checks whether `TimeWindow` has expired from the most recently detected time (`FlowEntry.Time`). When `TimeWindow` is expired, it recalculates $\texttt{est}_\texttt{T}$ in proportion to `TimeWindow` and compares it with `Threshold` to determine if the flow is heavy. The detected flow information and the corresponding action are sent to *OvS_system_call* for further flow management. *Rule_Matcher* then resets $\texttt{FlowRecord.est}_\texttt{T}$ and updates `FlowRecord.time` with the current time for the next round of detection.

3.5 Implementation

To show the feasibility of RFlow$^+$, we prototyped an SDN-based WLAN flow-level monitoring and management system. First, we describe our testbed, including settings, packet monitoring, traffic shaping, and performance. Second, we discuss the monitoring scenario of our testbed in terms of proactive and reactive flows. Last, we present two use cases, namely, 1) monitoring and limiting short-term/long-term heavy users and 2) flow table overflow detection.

3.5.1 Testbed Description

As shown in Fig. 24, we constructed our own testbed with three OpenVSwitches (*i.e.* OvS-WiFi, OvS-fast, and OvS-slow) in off-the-shelf AP hardware (TP-Link C7 AC1750 v2.0), which ran OpenVSwtich (2.3.90) on top of OpenWrt (15.05, Chaos Calmer). TP-Link AC1750 is a Qualcomm Atheros QCA9558 platform based wireless router that has 720 MHz CPU computing power and equips 128 MB DDR2 RAM with 16 MB additional flash memory. Basically, OvS-fast and OvS-slow worked as gateway interfaces in the LAN Zone and obtained Internet access from the WAN Zone. The bandwidth of these two interfaces was configured with different rates (OvS-fast at 72 Mbps and OvS-slow at 7.2 Mbps[3]) using a QoS software called `qos-script` [69], and they were both connected to the OvS-WiFi. To provide Internet access, the wireless interface `wlan1` was connected to the OvS-WiFi. The number of interfaces of an AP is relevant to the number of OVS instances in our RFlow+. All the physical wireless interfaces (*e.g.* 2.4GHz and 5.0 GHz WLAN NICs) can be attached to the OvS-WiFi at the same time. Therefore, all the network flows (traffic) are managed by OvS-WiFi, and are monitored RFlow+ local agent. Overall, the OvS-WiFi works as a central switch that communicates with an SDN controller and forwards packets referring to the flow definition in the flow tables. In the control layer, we used the OpenDaylight (Helium-SR4) as an SDN controller.

Traffic Shaping. We note that OvS-fast and OvS-slow are not acting as general OvS bridges but are configured as virtual interfaces in the LAN zone. They are both responsible for forwarding traffic between OvS-WiFi and WAN Zone; however, the benign users' traffic will be forwarded to the fast interface to enjoy a full bandwidth service, whereas the malicious users' flow (once be detected by our local agent) will be redirected to the slow interface. The redirecting of malicious flows is done by assigning flow entries with a relatively higher priority than the benign flow entries.

[3]Please note that the shaping rate is configurable by changing a parameter of `qos-script`.

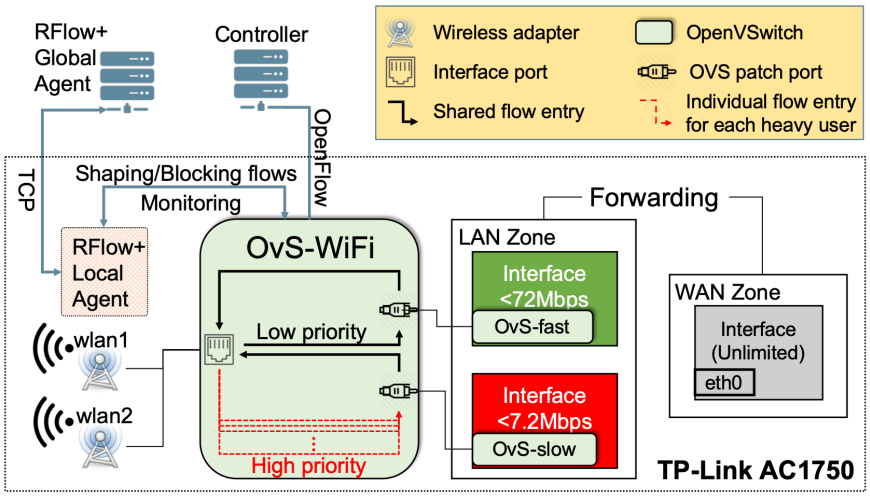


Figure 24: Testbed configuration: our AP consists of three OvS, namely OvS-WiFi, OvS-fast and OvS-slow. OvS-fast is a full bandwidth interface for normal users and OvS-slow is a bandwidth limited interface for shaping abnormal users. RFlow$^+$ local agent monitors OvS-WIFI and redirecting abnormal users' traffic to OvS-slow by defining high priority flows in OvS-WiFi.

Packet Monitoring. Traffic monitoring of our local agent is done by snipping the virtual switch (OvS-WiFi) using `Libpcap-1.5.3-1` [70]. Operations of RFlow$^+$ are totally independent of the OvS packet process pipeline to prevent the packet forwarding delay. For every packet that goes through the OvS-WiFi, the local agent extracts the flow information from its packet header and then performs counting tasks using our sketch pipeline. To redirect a malicious flow to the bandwidth-limited interface, RFlow$^+$ local agent defines a flow in OvS-WiFi by calling a system call of OvS.

Performance. In case there is only one user device, one can use the whole bandwidth. If there are multiple user devices connected to a single AP, they will fall in a race condition which follows the Probe Request and Response mechanism defined by 802.11 standard [37]; however, the total peak rate should be 72 Mbps considering our local agent does not present a notable CPU overhead, as shown in Fig. 25.

3.5.2 Proactive Flows and Reactive Flows

Today's OpenFlow networks are rarely used in a reactive mode due to scalability and performance reasons. However, operating an SDN-enabled device in a proactive mode is impractical because of the mobility and dynamics of WLAN users. Even though a predefined high-level (*i.e.* interface to interface) flow can provide reachability to the Internet, but we

will lose control of the network because fine-grained flow-level (*e.g.* layer 3 or 4) statistics are missing.

Native-OF. To achieve the fine-grained statistics, we need a reactive control plane to define all layer 3 or 4 flows for both directions (*i.e.* ingress and egress), which lead to an exponential increase of flow entries in an SDN-enabled AP. Even worse, recent WLAN measurement studies have reported the huge number of devices per AP at different places and different time periods [5, 27, 77]. Since a large number of flow definitions causes a heavy burden on the OvS (*i.e.* the AP in our setting), it is infeasible to operate native-OF in the reactive mode.

RFlow$^+$ Local Agent. To better cope with this, we suggest defining three proactive flows at the OvS port level (solid black lines in the OvS-WiFi). These high-level proactive flows are provided to benign users for acquiring Internet service. Instead of using the statistics collection function that is provided by native-OF, we use our sketch pipeline to perform the per-flow monitoring and detection independently. As described in Section 3.4, the sketch pipeline can monitor all packets passing through the OvS-WiFi and accumulate the statistics in the Local Flow Record Table. By combining with the predefined rules that defined by a northbound application, a high priority flow (red dash lines in the OvS-WiFi) is defined reactively when a benign user is identified as the malicious user (*e.g.* heavy hitter). Since the individual flow for a heavy user has a relatively higher priority than the shared proactive flow, the heavy user's traffic is redirected to the OvS-slow gateway with limited bandwidth (7.2 Mbps instead of 72 Mbps).

3.5.3 Use Cases

Deploying an SDN-enabled WLAN device in a wild environment is changeable due the threats it has to face. Among them, the most of critical threat is the resource exhaust attack which targets either on the bandwidth or the local flow table. The former attack can be detected by heavy user detection, which can be classified into two types according to the detection period: short-term or long-term. The time frame for a long-term heavy user might be a day, week, month, or longer, and that for the short-term user might be 500 ms, 50 ms, or less. The later attack can be captured by counting the cardinality of user flows. Among these detections, the short-term heavy user detection and flow table overflow detection are required to be efficient so that the treatments can be performed on-site and in a timely manner.

Long-term Heavy User Detection. Obviously, the time frame for a given quota should be longer than the statistics update period to the RFlow$^+$ global agent (in case of OpendayLight,

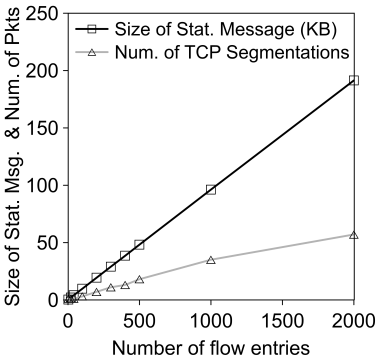


Figure 25: Network overhead of Native-OF varying the number of flows.

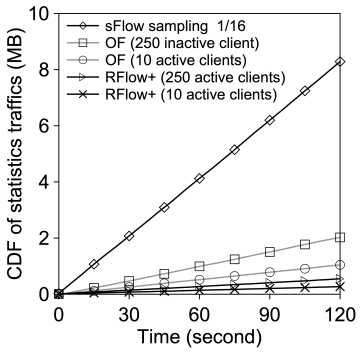


Figure 26: Network overhead comparison among RFlow$^+$, OpenFlow and sFlow over time.

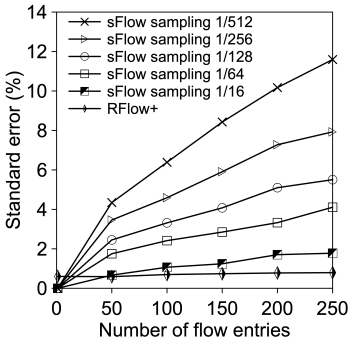


Figure 27: Accuracy comparison between RFlow$^+$ and sFlow varying the number of flows.

a default of three seconds). Also, the threshold for classifying a heavy user should be large enough so that normal users should not easily reach the limit in a given term (false positives). Even though both RFlow$^+$ and native-OF could be used to detect long-term heavy users at server-side for performing eventual actions (usually policies), native-OF costs additional storage for accumulating statistics and significant network overheads caused by the substantially larger number of flows (compared to RFlow$^+$) and customized flows.

Short-term Heavy User Detection. The time frames for the short-term detections (say, burstiness, or MAC flooding) should be shorter than periodic statistics updates; otherwise, the detections may fail to deliver the statistics to the northbound application in time for it to perform immediate actions/treatments. Therefore, it is desirable to measure the short-term heavy user locally (*i.e. on-site*) and execute immediate actions according to predefined rules. The RCC provides a good real-time estimation performance for detecting a short-term heavy user using a very small amount of memory. Using RCC, RFlow$^+$ is designed to execute locally the predefined immediate action rules in the RFlow$^+$ local agent.

Flow Table Overflow Detection. As discussed earlier, the proactive (defined at OvS port-level) flows to reduce the burden of the controller's southbound link but lose the capacity of collecting per-flow statistics (layers 2–4). As an alternative, RFlow$^+$ provides generic statistics using RCC. However, RCC alone cannot provide detection capabilities for DDoS-like attacks. For those attacks, RCSE guarantees an online performance to detect them without sacrificing accuracy. It is crucial for the operation and performance of RCSE to be implemented locally because of two reasons. First, it is important to do so because RFlow$^+$ cannot observe `PacketIn` events anymore due to the proactive flows. The other reason is

that recognizing and sending mice flows to the collector imposes a huge amount of overhead on both the switches and the collector.

3.5.4 Parameters

As shown in Nyang *et al.*'s work [66], 8 to 32 bits are sufficient for the virtual vector size of RCC. For RFlow$^+$, we use a unit of 8-bit virtual vectors, as this value obtained the best counting accuracy. We further confined the writing space of virtual vectors as a 32-bit word, which was the CPU word size of the testing devices. To evaluate the per-flow counting accuracy of RCC, we allocated 2 MB for each layer to perform the per-flow measurement. In each layer, we allocated 0.5 MB to the RCC sketch (*i.e.* RCC-L2, RCC-L3, and RCC-L4) and 1.5 MB for each layer's local flow record table (*i.e.* Table-L2, Table-L3, and Table-L4). Thus, in total, 6 MB of memory was allocated to RCCs. To evaluate the accuracy of RCSE, we varied the memory size of the sketch from 32 KB to 512 KB for the layer-3 flow table overflow detection task. To evaluate the overall CPU overhead of RFlow$^+$, we also allocated 512 KB for each layer's sketch. Thus, 1.5 MB was allocated to RCSEs. The size of the virtual vector of RCSE was varied from 16 bits to 64 bits to represent the threshold of detection of 44 to 266.

3.6 Evaluation

In this section, we show the performance of RFlow$^+$ in terms of network overhead, CPU overhead, and accuracy. First, we show the OpenFlow is lacking scalability and compare RFlow$^+$ with two feasible per-flow monitoring solutions, namely native-OF and sFlow. Second, we evaluate the CPU overhead of our framework using a real WLAN device. Third, we show the accuracy of our per-flow measurement algorithm for different time periods (*i.e.* 50 ms and one week). Fourth, we show the performance of our multi-tenant cardinality measurement algorithm in terms of false positive and false negative. Last, we present an application that heavy hitter quarantine to show the effectiveness of our framework.

3.6.1 Network Overhead

In the SDN environment, a controller sends `OF_FLOW_STAT_REQUEST` messages periodically to the APs (OvSs) for the collection of per-flow statistics. Then, each OvS packs the statistics and the definition of flow entries, which is defined in the userspace flow tables and generates an `OF_FLOW_STAT_REPLY` message for replying. For the experiment, the statistics updating period was set to three seconds, and the amount of traffic generated by `OF_FLOW_STAT_REQUEST`

and `UpdateTable` was measured for two minutes in our testbed. Fig. 25 shows the size of the `OF_FLOW_STAT_REPLY` messages and the number of TCP segments according to increments in the flow entry number defined in the flow tables. When 100 flows were defined in the OvS flow table, the size of a `OF_FLOW_STAT_REPLY` message was 9.78 KB, and it was segmented into three TCP packets.

For 2,000 flows, the size was increased to 191 KB and 57 packets. In reality, more than 57 segmented packets were sent owing to the packet loss, and ACKs were also transferred to the collector in a collecting period. We claim that assuming 2,000 (or more) flows is reasonable, even in a small scale environment (*e.g.* wireless LAN) because defining flows in the IP layer or higher, like `IP&PORT`, is necessary for various applications (*e.g.* billing or traffic analysis in various layers). In contrast, with the same rate of periodic statistics updates, RFlow$^+$'s message size was not proportional to the number of flows, as only active flows in a collecting period are packed into an update message.

Comparison with Native-OF. Fig. 26 was obtained by conducting experiments in our testbed. We use 250 clients to refer to the scenario that 500 flow entries are defined in the userspace table. Since the number of flows generated by a client is unpredictable, thus, we assume each client triggers two flows' defining when using native OF's reactive mode. In fact, to have generic (fine-grained) flow statistics with native OF, the total number flows should be much larger than 500 flows with a much smaller number of clients. When there were 250 clients (500 flows for both directions) in the network, the amount of traffic was the same irrespective of the liveness of the flows. Because RFlow$^+$ updates only the flow statistics that have been changed in an updating period, the amount of traffic was only in proportion to the number of active users, which was much less than that of native-OF. The amount of traffic required for native-OF is 64.4 times more than that of RFlow$^+$ with 10 active clients, and 7.68 times more for 250 active clients.

Comparison with sFlow. Fig. 27 shows RFlow$^+$'s and sFlow's standard error when the number of flows and sampling rate are varied. The dataset used for this experiment was a one minute network trace from CAIDA. The traffic was collected at the Equinix Chicago data center from 13:10–13:11 on the November 21, 2013 [**?**]. Even in a short time, the backbone generated 40.5 million packets from 264 K layer-3 flows, where the flow sizes are ranged from 1 to 3,327,267 packets. To get higher accuracy, a higher sampling rate is needed, as confirmed in the figure. In addition, for sFlow to achieve an accuracy that is comparable with RFlow$^+$, the sampling rate should be at least 1/16 in Fig. 27, but Fig. 26 shows that the network overhead of sFlow with a 1/16 sampling rate is significantly higher than that of RFlow$^+$. The amount of traffic required for sFlow is 276.2 times more than that of RFlow$^+$ with 10 active clients, and 33.1 times more for 250 active clients.

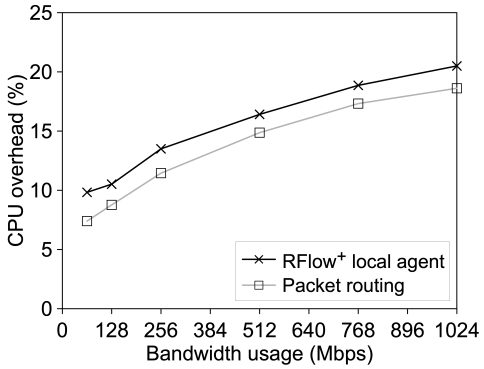
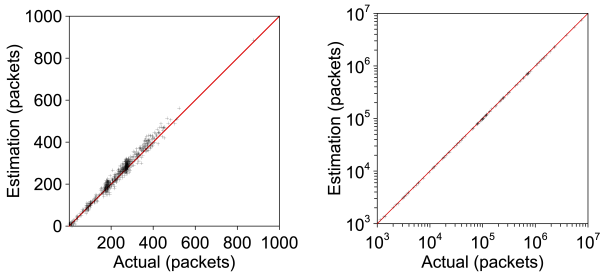


Figure 28: Overall CPU overhead of RFlow$^+$ local agent compares to that of packet routing only scenario by varying bandwidth utilization.

3.6.2 CPU Overhead

To evaluate the CPU overhead of RFlow$^+$, we enabled all sketches of the pipeline to perform layer-2 to layer-4 measurement tasks at the same time. Then, we measured the CPU usage by varying the bandwidth utilization. To generate constant network traffic that goes through the OvS, we used `iperf-3.1.3` [38] for generating UDP traffic from a wired LAN port to a wireless client. By doing so, the constant network traffic can be observed and measured by RFlow$^+$'s local agent, regardless of unstable wireless communication. Please note that our wired client generates a single but stable network flow because we focus only on finding out the additional CPU overhead that is presented by our local agent. Moreover, the entropy of the network traffic does not affect the computational complexity of the monitoring process (*i.e.* sketch pipeline). Fig. 28 shows a comparison of the CPU usage in different scenarios: with RFlow$^+$ local agent (blue solid line) and without RFlow$^+$ local agent (black dash line). In this experiment, the average CPU usage was measured under stable traffic for 30 seconds. As shown, the overall CPU usage linearly increases according to the increment of the bandwidth utilization. Starting from 9.82% (10 Mbps), RFlow$^+$ presents 20.5% CPU overhead when the bandwidth reaches the maximum bandwidth (*i.e.* 1 Gbps). To examine the CPU overhead presented by RFlow$^+$'s local agent, we disabled all of RFlow$^+$'s functions and repeated the experiments. As a result, the CPU overhead was only slightly lower than the overall CPU overhead, which was ranging between 7.8% (10 Mbps) and 18.61% (1 Gbps). Overall, we report that the CPU overhead in this experiment is caused mainly by the packet processing. That is, RFlow$^+$ added only a small amount of overhead, which proves the possibility of online processing. We note that our testbed never suffers from the maximum bandwidth degradation when running RFlow$^+$'s local agent.



(a) Short-term measurement (50 ms)

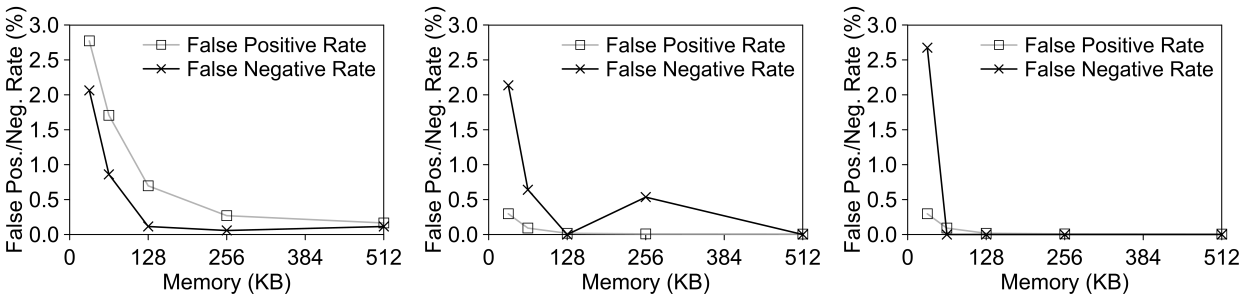
(b) Long-term measurement (a week)

Figure 29: Estimation accuracy of RFlow$^+$. Each point stands for a user flow, closer point to $y = x$ means more accurate estimation. RFlow$^+$ achieved 5% standard error in 50 ms period measurement for flows that less than 1000 packets. For a week period, RFlow$^+$ provided around 1% standard error for user flows that from 10 APs installed on our campus.

3.6.3 Accuracy of RFlow$^+$

Short-term Measurement. To test the accuracy of RFlow$^+$ for a 50 ms period, we played a 3 min video on YouTube at the 4K quality to generate traffic. The estimated and actual packet numbers for each 50 ms were compared. Fig. 29(a) shows the estimated number (Y-axis) collected by RFlow$^+$ as a function of ground-truth (actual packet number). The closer a point is to the guideline y = x, the more accurate the estimation is. The standard error for the short-term measurement by RFlow$^+$ is 5% in the estimation range of $0\tilde{1},000$ packets, which means that the measurement is underestimated or overestimated only by 25 packets for a 500 packet flow. As far as we know, no other monitoring system provides this level of accuracy for short-term monitoring.

Long-term Measurement. To evaluate RFlow$^+$'s accuracy for long term monitoring, we installed RFlow$^+$ on ten off-the-shelf TP-Link APs and deployed them on our campus. The 10 APs provided free Internet service for students on campus during summer vacation. Fig. 29(b) shows the estimated number (Y-axis) collected by RFlow$^+$ for a week. For the long-term measurement, we compared the packet number estimated by with the ground-truth. As shown in the figure, the estimations lie on the guideline y = x, which verifies that RFlow$^+$ provides high precision for long-term monitoring. The standard error for the long-term measurement by RFlow$^+$ is around 1% while consuming an extremely small amount of network overhead.



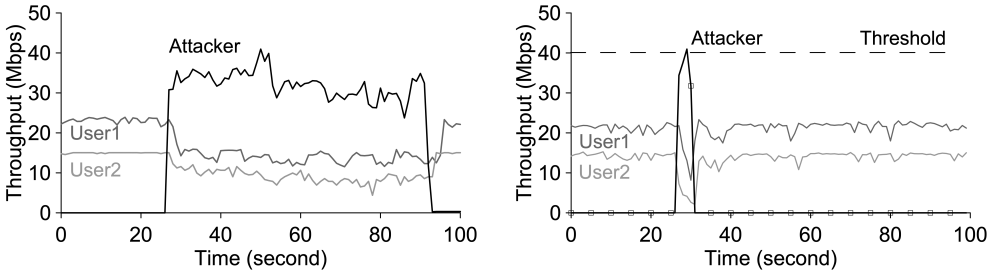
(a) Virtual vector size 16 (b) Virtual vector size 32 (c) Virtual vector size 64

Figure 30: Results of flow table overflow detection using RCSE. The virtual vector size is varied from 16 to 64 for different detection thresholds. For each virtual vector size, we varied the memory size of RCSE to show the accuracy in terms of false positive rate and false negative rate.

3.6.4 Flow Table Overflow Detection

To evaluate the accuracy of the RCSE, we use the one-hour CAIDA dataset [**?**], which contains around 1.3 million layer-3 flows to simulate the source IP-based flow table overflow detection. Fig. 30 shows the false positive rate and the false negative rate of RCSE by varying the memory usage and the virtual vector size. In those experiments, we used three thresholds by varying the virtual vector size of RCSE from 16 bits to 64 bits. At the same time, the memory usage ranged from 32 KB to 512 KB.

Fig. 30(a) shows the result when the virtual vector size was 16 bits of which threshold is 44 according to the formula of LC [**?**]. Accordingly, a report of a source IP means that the IP address sent packets to more than 44 IP addresses as destinations. As shown, the 32 KB memory size results in a high detection error rate in terms of both the false positive rate (FPR≈2.77%) and the false negative rate (FNR≈2.06%). Subsequently, RCSE requires 128 KB to reduce both false positive and negative rates under 1% (*i.e.* FPR≈0.69% and FNR≈0.11%). Finally, both the false positive and negative rates become extremely small when the memory size is 512 KB (*i.e.* FPR≈0.17% and FNR≈0.11%). Fig. 30(b) shows the result when using the 32-bit virtual vector, where the corresponding threshold is 110. As shown in this figure, to maintain both false positive and negative rates under 1%, the 32-bit RCSE requires only 64 KB, which is half the amount of the memory for the 16-bit virtual vector. As such, the false positive rates are extremely low (*i.e.* 0%-0.12%) after increasing the memory size to 128 KB. The false negative rates become 0% with 128 KB. Finally, Fig. 30(c) shows the result for the 64-bit virtual vector with a threshold of 266. As shown, RCSE requires only 64 KB to achieve 0.09% of FPR and 0% of FNR.



(a) Native-OF

(b) RFlow$^+$ with predefined attacker quarantine

Figure 31: Effectiveness of the MAC flooding attacker quarantine. Without RFlow$^+$, normal users' traffic was degraded by the attacker's flooding traffic. On the contrary, RFlow$^+$ can quarantine the attacker's traffic in a short period so that normal users' traffic was recovered immediately.

Overall, the accuracy of RCSE is proportional to the memory space when the size of the virtual vector (or threshold) is fixed. Moreover, when we have to guarantee a certain detection error rate, we can adjust both the threshold and memory size for achieving better performance. The overall results show that RCSE is able to guarantee an extremely low error rate ($< 1\%$) with small memory space and provides an adjustable threshold to monitor the attacks on demand.

3.6.5 Effectiveness of Heavy Hitter Quarantine

To test the effectiveness of the short-term monitoring and enforcement of predefined immediate action in the local agent, we artificially created two normal users who sent user datagram protocol (UDP) packets and consumed about 15–25 Mbps of bandwidth, and one attacker with `macof` [59], who sent randomly generated UDP packets and consumed more than 35 Mbps of bandwidth to cause MAC flooding by filling in the AP's content addressable memory (CAM) table and neutralizing its MAC learning. In Fig. 31(a), native-OF shows that normal users experienced significant throughput degradation owing to the traffic bursts caused by the attacker. Before 25s, only the normal users send packets in a low-fluctuating bandwidth, but right after 25s, the attacker starts to send a massive number of packets. This bandwidth-hogging creates a heavy load over the router's saturation bandwidth. As a result, the normal users' bandwidth started to fluctuate severely. In Fig 31(b), however, RFlow$^+$'s local agent continued to monitor every flow and penalized the attacker hogging bandwidth with a quarantine. When the RFlow$^+$ local agent detects that the trial attacker exceeds a pre-defined threshold, the agent quarantines the attacker's flow to suppress ruthless sending

so that the normal users can recover from the degraded bandwidth utilization (returning to normal).

3.7 Conclusion

In this paper, we presented RFlow$^+$, a novel SDN-based WLAN flow-level monitoring and management framework for separately handling immediate action for short-term (*e.g.* 50 ms) monitoring results, and eventual action for long-term (*e.g.* one month) results. We also discussed the potential threat of an SDN-based WLAN device when deploying in the wild environment. To address the threat, we propose an online decodable flow table overflow algorithm to prevent resource exhaust in a timely manner. Through extensive experiments, we showed our algorithm is highly accurate and feasible in resource-constrained devices (*i.e.* WLAN router). Further, we integrated our sketches with pipeline design, prototyped our framework in an off-the-shelf device, and deployed our devices on our campus. To show the feasibility, we compared the accuracy and network overhead of RFlow$^+$ with existing solutions (*i.e.* OpenFlow and sFlow) and verified the practicality of RFlow$^+$ by showing the effectiveness of the detection and quarantine of a MAC flooding (bandwidth-hogging) attacker.

4 SketchFlow: Per-Flow Systematic Sampling UsingSketch Saturation Event

4.1 Introduction

The simple random sampling (SRS) has played an important role in network traffic measurement, resulting in standards such as Sampled NetFlow [12] and sFlow [85]. For instance, the Sampled NetFlow samples packets to reduce the CPU overhead of switches to prevent delay in routing decisions. sFlow uses simple random sampling to reduce meta-data transmission over the network. Sampling has been comprehensively studied, since the work of Claffy *et al.* [14], which uses sampling for gathering network statistics. As an alternative solution, however, sketches have been introduced by Morris [62] and Flajolet *et al.* [25]. Since then, many works have been conducted to enhance sketches' accuracy while reducing their overhead [24, 49, 66, 93]. A comparative study of sampling and sketches has been done by Tune *et al.* [92].

Sampling is a practical solution in many areas, such as network measurement and high-volume data analysis (categories of sampling are shown in Fig. 32). As such, it has played a

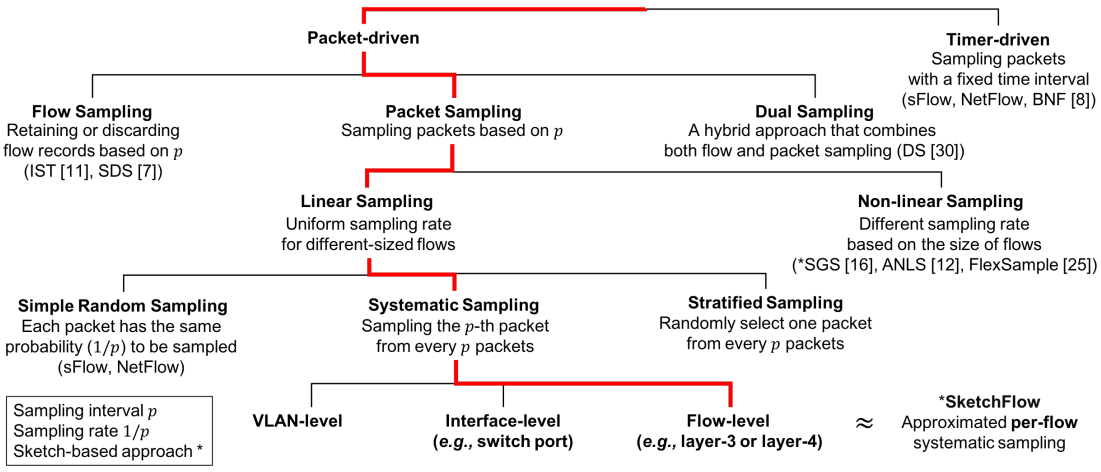


Figure 32: Design space of SketchFlow.

significant role as a filter to reduce the burden on the flow record table (*e.g.* in NetFlow) and to lessen the network bandwidth overhead (*e.g.* in sFlow). Therefore, maintaining a stable task reduction rate is a crucial part of evaluating sampling algorithms, where the reduction of the influx of elements is determined by the *sampling rate*, which also leads to the well-known trade-off between accuracy and overhead. A large sampling rate (*e.g.* 1/10) achieves high accuracy by conducting fine-grained sampling by obtaining samples more frequently. On the contrary, a small sampling rate (*e.g.* $1/10,000$) provides coarse-grained samples (*i.e.* relatively low accuracy), but fewer samples are taken. To provide a better trade-off, many sampling strategies have been proposed. Claffy *et al.* [14] showed that timer-driven sampling does not perform as well as event-driven (or packet-driven) sampling. Among packet-driven sampling methods, most research works are on packet sampling, but flow thinning or flow sampling has been shown to be better in terms of its accuracy [34]. However, it heavily relies on additional information, such as TCP SYN/SEQ signals. That means the sampling is not general enough to be used for other purposes such as UDP traffic measurement–QUIC (Quick UDP Internet Connections) has occupied 7% of the global traffic in 2016 (and more than 7.8% as of late 2018) [79]. Moreover, such an approach has to manage flow labels in a hash table, which is another challenge.

Packet sampling is categorized into linear and non-linear sampling, per Fig. 32. The linear sampling is featured by uniformly sampling $1/p$ packets of a data stream, where p is the sampling interval and $1/p$ is the sampling rate. According to Claffy *et al.* [14], the simple random sampling, stratified sampling, and systematic sampling can be applied as sampling strategies. Recent works have focused on how to apply a non-linear sampling rate according to the flow size [35,51,76], where mouse flows get sampled more often and elephant flows less often using a non-linear function based on the flow size. On the downside, the non-linearity

in the sampling rate substantially increases the overhead by sampling small flows heavily to guarantee the accuracy for a traffic distribution.

Sketches are compact data structures that use probabilistic counters for approximate estimation of spectral densities of flows [17,50,54–56,66]. Sketch-based algorithms have been shown more accurate in estimation than sampling approaches while using a small amount of memory. The higher accuracy of sketches is owing to its per-flow nature of estimation. However, research on sketch-based estimation has mainly focused on the sketch itself: the very nature of sketch to use only a small amount of memory prohibits it from being used for processing large scale data. More specifically, once a sketch is saturated, it cannot count at all. Consequently, sketch-based measurement algorithms have been used in a limited way such as for anomaly detection (*e.g.* heavy hitter, super spreader, etc.) within a short time frame [36, 55, 57, 98]. Also, decoding a sketch is computational heavy, thus cannot be done in data-plane, and thus sketches usually are delivered to a server with enough computing power for decoding, which inevitably introduces a control loop delay.

Our goal is to design a new sketch-based sampling algorithm, called SketchFlow, to provide a better trade-off between accuracy and overhead for a given sampling rate of $1/p$. SketchFlow performs an approximated systematic sampling for fine-grained flows (*e.g.* layer-4 flows) independently. As a result, almost exactly $1/p$ packets from each and every flow will be sampled. This property is in contrast to SRS, in which the sampling rate across different flows in a data stream is not guaranteed. SketchFlow provides a high estimation accuracy, processes high-speed data in real-time, and is general enough to be used for many estimation purposes without any application-specific information. The core idea of SketchFlow is to recognize a sketch saturation event for a flow and sample only the triggering packets. The saturated sketch for the flow is reset so that it can be reused. Therefore, SketchFlow can be seen as a sampler as well as a sketch. SketchFlow, however, does not work alone as a sketch measuring the whole data stream, but as a general sampler to NetFlow and sFlow.

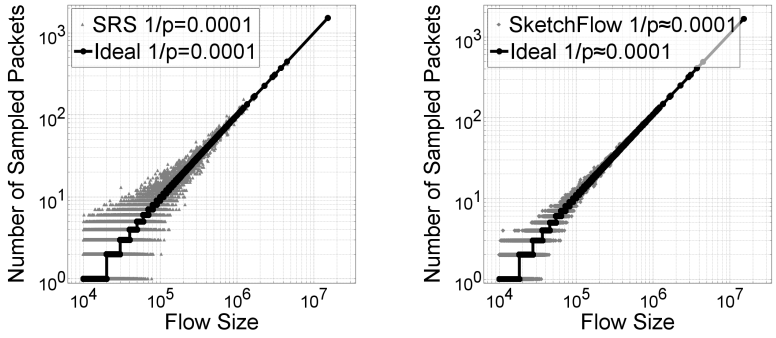
In summary, our contributions in this paper are as follows: 1) We introduce the new notion of per-flow systematic packet sampling for a precise sampling. See Fig. 32 for how our contribution fits within the literature. 2) We propose a new framework using the per-flow sketch saturation event as a sampling signal of the flow, whereby only a signaling packet is sampled from the flow, and the saturated sketch is emptied for the next round sampling. This use of a sketch as a sampler is new in the sense that a per-flow sketch now works as a per-flow systematic sampler, and the sketch saturation is not any more an issue. We note, however, that a sketch is an approximate per-flow counter thus a sampling algorithm under the framework is only an approximate per-flow systematic sampler. A new instance can be designed using any better sketch when available. 3) We realize an approximate version of per-flow systematic packet sampling called SketchFlow. For this purpose, a new

per-flow sketch algorithm is presented, which can encode and decode flows in real-time. Multi-layer sketch design is applied for scalable sampling. 4) We demonstrate SketchFlow's performance in terms of the stable sampling rate, accuracy, and overhead using real-world datasets, including a backbone network trace, hard disk I/O trace, and Twitter dataset.

4.2 Motivation: Flow-aware vs. Flow-oblivious Sampling

The bottleneck of NetFlow is the processing capacity for the local table, and that of sFlow is the network capacity. To address the bottleneck, the widely-adopted simple random sampling (SRS) is used with a very small overhead. In theory, SRS guarantees each packet has an equal chance to be sampled. However, the general usage of SRS is for sampling over the interface or VLAN, which collects coarse samples without considering the individual fine-grained flows, such as a flow defined by the 5-tuple. Consequently, some flows are sampled more than the designated sampling rate, resulting in over-estimation, while others suffer from under-estimation. We note that, although the main purpose of traffic measurement is mostly to obtain per-flow statistics such as the spectral density of flow size and distribution, sampling has been applied to data streams aggregating all the flows, rather than individual flows. SRS samples packets with $1/p$ over the entire data stream, although it cannot guarantee the sampling rate to be $1/p$ for each flow. For per-flow statistics, however, the estimation accuracy is ideal when exactly f/p packets for each flow are sampled (See the solid lines in Fig. 33), where f is the flow size and $1/p$ is the sampling rate. If more or fewer packets than f/p are sampled for a flow, it leads to over- or under-estimation of the actual flow size, because the number of the sampled packets is multiplied by p to estimate f. Therefore, the best strategy is to keep the per-flow sampling rate identical across flows. To that end, we propose the *per-flow systematic packet sampling*, which is a method to sample every p-th packet within a flow, whereas the well-known packet-level systematic sampling is to sample every p-th packet over the entire data stream. Fig. 33(a) shows the number of sampled packets according to flow size for a given sampling rate. The sampling quality is captured by how close the grey dot (the number of actually-sampled packets) is from the solid line (the number of ideally-sampled packets) in this figure. Here, we see that the sampling quality of the flow-oblivious sampling, such as the simple random sampling (*i.e.* SRS), is much poorer than that of the per-flow systematic sampling (*i.e. ideal*), which is a flow-aware sampling algorithm.

The complexity of the per-flow systematic sampling problem is equivalent to the per-flow counting problem, which means we still have to pay a large amount of memory/computations for the flow table (*i.e.* fail to reduce the complexity). To address this issue, we propose a sketch saturation-driven per-flow systematic sampling framework. Our framework utilizes a



(a) Flow-oblivious sampling (b) Flow-aware sampling

Figure 33: Number of sampled packets compared to exact per-flow systematic sampling (*i.e.* ideal): the estimation of SketchFlow is more accurate than the simple random sampling (SRS).

sketch to reduce the complexity of the per-flow counting problem. The sketch in the framework, however, is used to estimate a sampling interval of a flow, rather than the flow size. Therefore, the sketch does not need to be large to hold the whole flows' total length, but it would be sufficient even when small because it holds only concurrent flows' sampling intervals and resets. When a packet arrives, the sketch encoding algorithm recognizes its flow to sketch individual flows on a small memory in real-time. When the sketch space is saturated, the triggering packet is sampled to a flow table (*e.g.* NetFlow) or to a collector (*e.g.* sFlow), and the saturated sketch is emptied for the next round sampling. One can build a per-flow systematic packet sampling algorithm easily from the generic framework by defining an online-encodable/decodable sketch algorithm. Since a sketch for per-flow estimation of the sampling interval has an approximate counting structure, a sampling algorithm from the framework is an approximate version of the per-flow systematic sampling, providing a very high accuracy in per-flow statistics while reducing the overhead (both tables and network bandwidth) by keeping the sampling rate consistent across flows.

SketchFlow is a concrete example of the framework. Fig. 33(b) illustrates the accuracy of SketchFlow. For each flow, the fraction of the sampled packet number over the flow size is almost equivalent to the sampling rate of $1/p$. Moreover, the variance of SketchFlow is much smaller than flow-oblivious sampling schemes (Fig. 33(a)). In addition, SketchFlow can provide mouse flow samples by stacking these flows to trigger sampling events (See section 4.5-D). To sum up, legacy SRS can be replaced by SketchFlow in many applications such as network monitoring (*e.g.* NetFlow and sFlow), big data analytics (*e.g.* PowerDrill [32]), and social network service data analysis (*e.g.* Twitter and Facebook) for better performance.

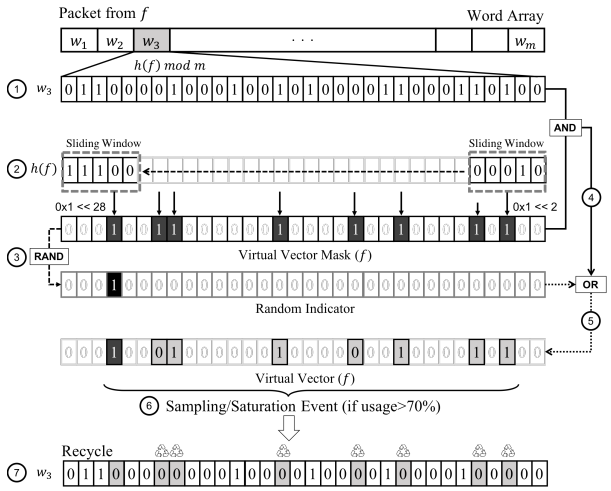


Figure 34: The overview of SketchFlow

4.3 Sketch-based Per-flow Systematic Sampling

We present SketchFlow, an instance of our framework using per-flow sketch to trigger per-flow sampling. SketchFlow is an approximate per-flow systematic sampling.

4.3.1 Encoding: Data Structure and Overview

Fig. 34 shows an overview of SketchFlow designed to perform an approximate per-flow sampling using a small amount of memory. We constructed SketchFlow using a word array, which is initialized to all 0's. When a flow f arrives, a word from the word array will be selected using the 5-tuple hash value $\mathtt{h}(f)$ (①) , and then s bits of the register (*i.e.* vector mask) are allocated to f according to the partial output (sliding window) of $\mathtt{h}(f)$ (②). The virtual vector is extracted by doing "Bitwise AND" between register and vector mask (④). For each packet from f, a randomization technique [66] is used for multiplicity counting. That is, one bit position of the vector is randomly flipped to 1 by ③-⑤. A sampling event of each flow is triggered when the usage of the vector exceeds its limit (*i.e.* vector saturation) (⑥), and then the estimated value of the average number of packets to saturate the vector becomes the sampling interval $\hat{p}$. In SketchFlow, Linear counting (LC) is used for volume estimation by $\hat{p} = -m \ln(V)$, where m is the number of bits (or memory size), and V is the fraction of 0's remaining in the vector. Our approach is consistent with the theory of LC, while inheriting its limitations—that is, LC guarantees accuracy only before 70% of a vector is exhausted [93]. After a sampling event, the vector is recycled (reset to 0) in anticipation of the next round sampling event (⑦). By doing so, the reduction ratio of each flow (equiv-

alently, the sampling rate) is approximately $1/\hat{p}$. Due to the constrained memory space, however, vectors have to be designed to share bit positions with one another (virtual vector hereafter), which brings about a major challenge, the noise owing to virtual vector collisions. That is, multiple concurrent flows fall in a race condition when claiming bits in shared bit positions. We carefully designed the sketch to take a noise-free approach by minimizing the race condition (See section 4.3.3).

4.3.2 Decoding: Sampling Trigger

Understanding the Saturation Event. A sampling event is triggered by the saturation of a virtual vector assigned to each flow, and the usage of virtual vector is monitored whenever a packet is encoded into it. The packet that triggers the saturation event of a vector (hereafter, signaling bit) is one that flips a 0's position to 1 and eventually causes more than 70% usage of the vector. We observed that the LC's formula could not be directly applied here, because it overestimates the number of packets encoded in the virtual vector. The key reason is that the last packet in a 70% marked vector is highly likely to remark the bit already marked in LC, whereas the signaling bit marks a fresh 0's bit in our sketch. We note that this estimation gap does not mean that LC is wrong, but event-driven sampling trigger (*i.e.* signaling bit) was not intended by LC.

Saturation Event-based Estimation (Sampling Interval). Here, we propose a new formula to calculate the estimation considering the saturation event, which is the basis of the real-time sampling.

Theorem 1. Considering the saturation event that triggers setting the $(s-z+1)$-th signaling bit in a virtual vector of size s, the sampling interval of a flow, $\hat{p}$ is calculated as follows:

$$\hat{p} = \frac{\ln V_z}{\ln\left(1-\frac{1}{s}\right)} + \left(\frac{1-V_{s-z}^{\tau}}{1-V_{s-z}} - \tau \cdot V_{s-z}^{\tau}\right), \tag{6}$$

where z (V_z) is the fraction of 0's in a virtual vector, τ is a positive constant, and s is the vector size. For convenience, we consider the first term as $f(z)$ and the second as $g(z)$.

Proof: The equation consists of two parts: the former modifies the LC's formula without truncating the minor terms, and the latter is the probabilistic expectation by considering the saturation event. Let n be the number of packets and $\hat{n}$ be the estimation of packets. In appendix A in [93], Wang *et al.* derived the mean of the random variable U_n which represents the number of 0's in the bit map, or a virtual vector. Let A_j be an event that the j-th bit is 0, and let 1_{A_j} be the corresponding indicator random variable. Then, since U_n is the number of 0's, $U_n = \sum_{j=1}^{s} 1_{A_j}$, where s is the size of the vector. Finally, per [93],

we have the following.

$$E(U_n) = \sum_{j=1}^{s} P(A_j) = s \cdot (1 - 1/s)^{\hat{n}}, \tag{7}$$

where $P(A_j)$ is the probability of A_j. Since the assignment of the bits is independent, $P(A_j) = (1 - 1/s)^{\hat{n}}$.

They approximate this equation to a convergence value when s and n go to infinity. However, for a more precise estimation, Nyang *et al.* [66] used the non-approximation estimation derived from the expectation of U_n, which is used as $f(z)$ for better accuracy, because the frequent accumulation of small estimation error can grow bigger. They obtained

$$V_z = (1 - 1/s)^{\hat{n}}, \tag{8}$$

where V_z is the fraction of 0's in the vector, that is, $E(U_n)/s$. And by taking the log, they deduced

$$\ln V_z = \hat{n} \cdot \ln (1 - 1/s). \tag{9}$$

We choose $\hat{n}$ as $f(z)$, because $\hat{n}$ is the estimation of packets when there are z zeros in the vector. Note that the first part $f(z)$ is not a cumulative sum of the second part $g(z)$ because z is the number of zeros before the signaling bit flips to 1's.

Let $g(z)$ be the expected number of packets required for saturation after a virtual vector state reaches to the state having $z - 1$ zero bits from z zero bits. We assume that $g(z)$ is the number of packets needed to make the event. This means that the first $g(z) - 1$ packets did not convert a new 0-bit to 1, and the last $g(z)$-th packet selects the 0-bit in the virtual vector. The probability of the former is V_{s-z}, the fraction of 1's in the virtual vector v, and the latter V_z, the fraction of 0's in v; namely, V_z equals $1 - V_{s-z}$. Since $g(z)$ must be a positive integer, we can expect $g(z)$ from 1 to some extent, τ. Therefore, we get the following expectation:

$$\begin{aligned} g(z) &= V_z + 2V_zV_{s-z} + 3V_zV_{s-z}^2 + \cdots + \tau V_zV_{s-z}^{\tau-1} \\ &= \sum_{i=1}^{\tau} \left(iV_zV_{s-z}^{i-1} \right) = \frac{1 - V_{s-z}^{\tau}}{1 - V_{s-z}} - \tau \cdot V_{s-z}^{\tau}. \end{aligned}$$

The last term in the above equation is obtained from $g(z) - V_{s-z} \cdot g(z)$. V_z in $g(z)$ is canceled out by dividing both sides by V_z; that is, $1 - V_{s-z}$. ■

In this paper, we set the number of trials (τ) to 8 because it has 95% of confidence on flipping a new 0's to 1's from having z zeros. A random variable $\mathcal{K}$ follows the binomial

distribution with parameters τ and V_z, where τ is the number of trials (or packets) and V_z is a probability that one packet make the saturation event.

Proof of Unbiased Sampling. The first term is unbiased when it is used to estimate the average number of packets per the virtual vector usage (See [93]). We use it to estimate the condition before the saturation event (*i.e.* $f(z)$). The second term is the expected number of packets (constant) that triggers the saturation event from the last condition (*i.e.* $g(z)$), which does not impact the variance of the entire formula.

Theorem 2. Assume that there is an initial virtual vector v for SketchFlow. We define the saturation event by the state transition from the state where the number of zeros in v is z to the state with $z-1$ (z is 30% of the virtual vector size when $s \geq 8$.). At the exact moment when the event has just occurred, SketchFlow's estimation of the number of packets needed to trigger an event is unbiased.

Proof: The first term of the estimation, $f(z)$, is the number of packets which is used to maintain z zeros in v. The expected value of $f(z)$, $E(f(z))$, is unbiased by LC's theory. Starting from the point when v has z zeros, the expected value of the number of packets for the saturation event is $E(g(z))$, which is also unbiased according to Theorem **??**. Therefore, SketchFlow's formula $f(z)+g(z)$ is unbiased, because $E\left(f(z)+g(z)\right) = E(f(z)) + E(g(z))$. ■

4.3.3 Estimation without Noise Reduction

In SketchFlow, a fixed virtual vector (of s bits) was "temporally" given to a flow for performing LC-like probabilistic counting. Thus, vectors of concurrent flows may partially or fully share bit positions, and bring about a race condition for the shared bit positions resulting in a virtual vector collision. We propose a noise-free approach to dramatically mitigate the virtual vector collision spatially and temporally. We also show that even when the noise occurs, SketchFlow can ignore the vector collision problem introducing the noise. For instance, once a specific flow triggers saturation event of the virtual vector, the flow takes all bits in the vector regardless of how many bits (or noises) were actually contributed by other flows, and it resets the vector. Our approach is tolerant to collision considering the following dispersion aspects:

Spatial Dispersion. Spatially, SketchFlow confines the virtual vector of flows within a word range (*i.e.* 32-bit or 64-bit), then distributes flows in the memory space (*i.e.* word array) uniformly. This greatly reduces the probability of collision of concurrent flows, when enough number of words for confinement are given. In a local view, SketchFlow uses a small size for virtual vectors, which is smaller than the word size. The probability of vector

collision within a s_w-bit word with respect to the size of vector (s_v) and the number of concurrent flows (n_f) is $p_{collision} = 1/\binom{s_w}{s_v \cdot n_f}$, where $p_{collision}$ decreases when s_v gets smaller. Both contribute to reducing spatial collision of virtual vectors of concurrent flows.

Temporal Dispersion. SketchFlow looks into a small timescale for TCP bursts. TCP usually sends a window of data in one or a few bursts and waits for ACKs, which causes a flow to be broken into many small subsets named flowlets. Sinha *et al.* [87] reported that the number of concurrent flowlets was much smaller than that of concurrent flows, which makes the probability of the spatial virtual vector collision even smaller in the smaller timescale. Moreover, the small vector size of SketchFlow increases the probability that the saturation events are triggered before the end of flowlets, which also reduces the probability of virtual vector collision in a temporal manner.

Worst Case. For the worst case, we can consider the situation where multiple concurrent flowlets share bit positions with each other. We claim that even without considering the noise by other concurrent flowlets, equation (6) is enough to decide whether the flow reaches the sampling interval or not. Whether two flows are mouse or elephants, probabilities of each flow to lose bits are the same in a sampling interval. This is because, during a sampling interval, two flows lose the concept of transmission rate but are only mixed in a random sequence in the buffer when concurrently arriving flowlets are loaded.

4.3.4 Scalable Sampling

As described in section 4.3, SketchFlow uses a virtual vector smaller than the size of the word. However, a 32-bit virtual vector cannot count over 40 (See Fig. 35(a)), which limits the minimum sampling rate. Increasing the confinement size does not help with scaling up the sampling interval but induces more memory read and write. To scale up the sampling interval, SketchFlow employs a "multi-layer" strategy where each layer of SketchFlow maintains an independent word array. Unlike other multi-layer sketch approaches that only scale up the retention capacity (*e.g.* [10]), SketchFlow provides an online decoding feature as well to help with the high-speed processing. Encoding the arriving packet starts from the lowest layer and climbs the layers depending on the saturation of the virtual vector. Repeatedly, the saturation from the lower layer is encoded into its upper layer following the same process of encoding. That is, the upper layer counts the saturation of its lower layer. Finally, the sampling event happens when the flow is saturated at the highest layer. All layers share the same hash value of a flow but run different random functions. The sampling interval of multi-layer SketchFlow is the multiplication of the sampling interval of each layer (See Fig. 35(b) for sampling interval by different layers). For 3-layer SketchFlow with an 8-bit virtual vector, the sampling interval is 9.764^3. Note that each layer can use different virtual

Algorithm 4: Encoding and Sampling Trigger

input: # of layer l, word array $w[l][]$, vector size s

```
1  forall Pkt_f do
2      h_f ← hash(Pkt_f);
3      w_v ← make_confined_vector(h_f);
4      for L = 0 to l − 1 do
5          w[L][h_f] ← w[L][h_f] | leave_one_bit_only(w_v);
6          /*Saturation event is triggered if usage > 70%*/;
7          if Popcount(w[L][h_f] & w_v) ≥ 0.7 × s then
8              w[L][h_f] ← w[L][h_f] & bitwiseNOT(w_v);
9              /*Sampling event is triggered in the last layer*/;
10             if L = l − 1 then
11                 Trigger a sampling event with flow f;
12             end
13         else
14             break;
15         end
16     end
17 end
```

vector sizes to achieve different sampling intervals on demand.

4.4 Implementation

4.4.1 Algorithm

SketchFlow's algorithm can be divided into encoding, sampling/saturation trigger, and multi-layer sampling phases.

Encoding. For each arriving packet of a flow f, SketchFlow computes the hash (h_f) of the 5-tuple extracted from the header (line 3). The h_f is used for two purposes. First, part of h_f is used to calculate the bit positions of the virtual vector in a word (line 4). By calling `make_confined_vector`(), we obtain a virtual vector bit mask in a word register (w_v) for one confinement in which only the bit positions of the virtual vector for f are set. Second, h_f is regarded as an index that determines in which word the virtual vector is confined among word arrays (line 5). Once w_v and $w[Layer][h_f]$ are ready, `leave_one_bit_only`() randomly selects one of the 1's position among w_v and "Bitwise OR" it with $w[Layer][h_f]$.

Sampling/Saturation Trigger. After several rounds of encoding, the virtual vector of f will be saturated (>70% usage). SketchFlow monitors the saturation of the vector after every encoding by counting the number of 1's using `Popcount`() [61] (line 7). Once the saturation threshold is reached, the bit positions will be reset to 0 (line 8), and the sampling/saturation event is triggered[4]. One sampled packet represents $\hat{p}$ packets in equation (6), which is a pre-decoded value and enables real-time sampling.

Multi-layer Sampling. To implement multi-layer SketchFlow, the encoding process is repeated (line 4-16) for each saturation event to the upper layer using the word array the layer belongs to. Eventually, the sampling event is triggered when the saturation events occur in the last layer (line 11-12). All layers share the hash value (h_f) and virtual vector mask(w_v) computed in the lowest layer to alleviate the computation.

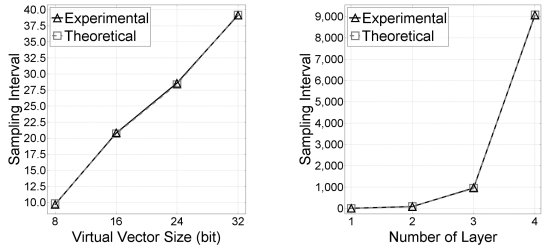
4.4.2 Parameter

The size of confinement of a virtual vector is selectable depending on the processor architecture (32 or 64 bits). The size of the virtual vector is recommended not to exceed half of the size of a word to reduce the probability of virtual vector collisions within a word. For memory usage, we recommend that the maximum possible number of virtual vectors that can be contained in a word array should be equivalent to the number of concurrent flows in a second for tolerant sampling. In our evaluation, we used a 110KB 32-bit word array per layer and an 8-bit virtual vector when performing the experiments using CAIDA trace because the maximum number of concurrent flows was ≈110K. We found that SketchFlow provides better accuracy than other sketch approaches even with a small memory usage (See section 4.5).

4.4.3 Performance Optimization

For real-time per-flow systematic sampling, we take several optimization efforts. 1) By careful design, SketchFlow requires only one conditional branch for each layer to trigger the sampling/saturation event. 2) For fast computation, SketchFlow marks the bit positions of the virtual vector in an empty register (line 3) so that encoding (line 5) and recycling (line 8) can be done in a single "Bitwise OR" and "Bitwise AND" operations. 3) Due to the confinement of a virtual vector, usage check of a virtual vector can be done using a built-in hardware population counting function (`Popcount`()) [61]. 4) Inspired by the implementation of the exact match cache (EMC) module of OpenvSwitch using DPDK [21], the hardware-

[4]If $s = 8$, a sampling event is triggered when 6 or more 0's positions are marked as 1's (*i.e.* $k = 6$), because 6 bits are 75% (>70%) of an 8-bit vector.



(a) Single-layer design (b) Multi-layer design

Figure 35: Theoretical and experimental sampling interval of SketchFlow.

based CRC checksum instruction of streaming SIMD extensions (SSE) [88] was used to calculate our 5-tuple hash function.

4.5 Evaluation

In this section, we use various metrics to evaluate SketchFlow. First, we compare our theoretically-estimated sampling interval with the experimental result to verify the sampling interval in equation (6). Also, we show the scalability of our multi-layer strategy in terms of the sampling interval. Second, we evaluate the overall performance of SketchFlow using CAIDA trace by varying the sampling rate and comparing SketchFlow with simple random sampling (sFlow [85]) and with a non-linear scheme (sketch guided sampling [51], SGS hereafter). Third, we discuss the overhead of SketchFlow. Lastly, we evaluated SketchFlow not only in the network traffic dataset [9] but also in the keyword ranking problem (Twitter dataset [53]) and in the hot block ranking problem (Disk I/O trace [64]) that has more complex data distribution.

4.5.1 Estimation Accuracy and Scalability

Fig. 35(a) shows sampling interval of SketchFlow by varying the virtual vector size. The Y-axis is the average number of packets to trigger a sampling/saturation event. We compared the estimated value of SketchFlow with the experimental results (1 million runs). As a result, our estimation is accurate regardless of the size of the virtual vector (error rate$<$0.07% for 8-bit). However, the growth rate is very slow, and so the counting capacity for a 32-bit virtual vector cannot go over 40 packets (Fig. 35(a)). With the multi-layer strategy, the counting capacity exponentially increased, as shown in Fig. 35(b). Using an 8-bit virtual vector for 4-layer SketchFlow which equally assigned 32 bits for each flow, the counting capacity dramatically increased to reach around 9,088. Note that to achieve the equivalent counting

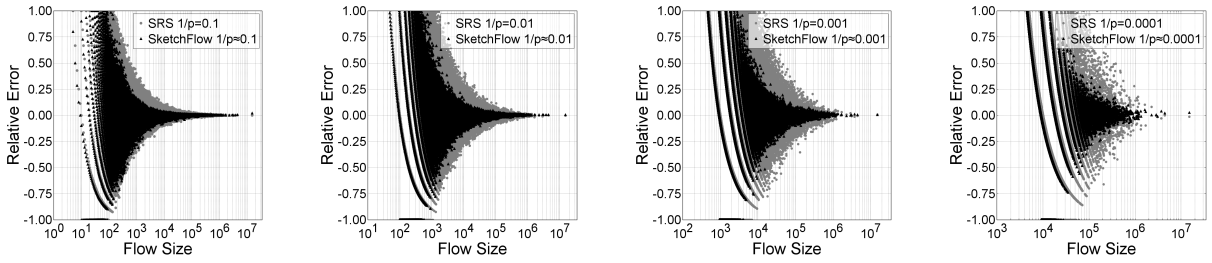


Figure 36: CAIDA trace: Relative error of independent flows of SketchFlow and SRS. Each point stands for each flow. To see how accurate each scheme is, check how close the point is to $y = 0$. Multi-layer SketchFlow was used to approximate sampling rates 0.01-0.0001 (left to right), respectively. Each layer was assigned with a 110KB 32-bit word array, and 8-bit virtual vector was used for all experiments. No memory usage is required by SRS. CAIDA trace contains ≈2 billion packets and ≈95 million L4 flows.

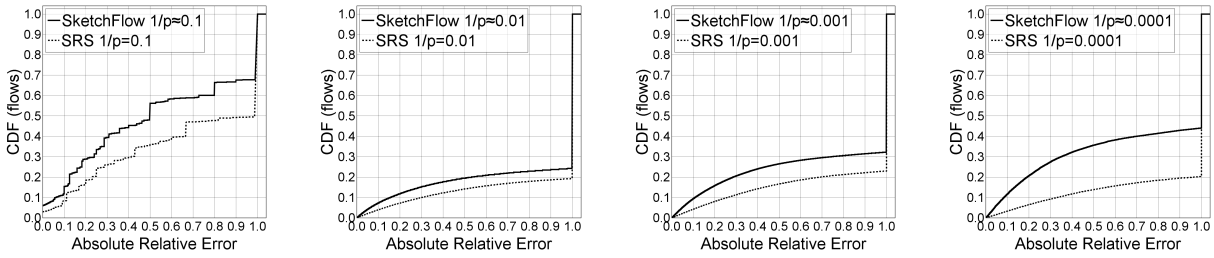


Figure 37: CAIDA trace: CDF of flow-level relative error of SketchFlow and SRS. The overall accuracy of SketchFlow is better than SRS.

capacity without the multi-layer strategy, thousands of bits are needed for a virtual vector. Furthermore, hundreds of memory accesses are required to decode it, which is unacceptable for online sampling. In the multi-layer mode, SketchFlow needs only one memory access for each layer.

4.5.2 SketchFlow vs. Linear Sampling Approach (SRS)

Per-flow Accuracy. For our baseline, we compared SketchFlow with SRS using the CAIDA trace. The implementation of SRS followed the way used in sFlow. To achieve the same sampling rate as SRS, SketchFlow approximated the sampling rate using the multi-layer strategy where each layer used 8-bit virtual vector. The approximated sampling rates of SketchFlow are 1/9.764 (L1), 1/95.328 (L2), 1/930.750 (L3) and 1/9087.749 (L4), respectively. In SketchFlow, each layer was assigned with a 110KB 32-bit word array so that the maximum possible number of virtual vectors without collision should be equivalent to the

Table 3: Flow Thinning Performance: Higher is better

Sampling Rate		0.1	0.01	0.001	0.0001
SketchFlow	precision	0.414	0.174	0.240	0.293
	recall	0.931	0.950	0.959	0.954
SRS	precision	0.408	0.161	0.201	0.159
	recall	0.916	0.960	0.921	0.923

Table 4: Packet Thinning Performance

Sampling Rate	SketchFlow		SRS	
	samples	ratio	samples	ratio
0.1	198,322,728	0.10156	195,274,392	0.10000
0.01	19,973,488	0.01023	19,531,764	0.01000
0.001	1,964,032	0.00101	1,952,120	0.00100
0.0001	154,041	0.00008	195,865	0.00010

maximum concurrent flows of CAIDA trace in a second. No memory usage is required by SRS. Fig. 36 presents the relative error of SketchFlow and SRS varying sampling rates, where SketchFlow's estimation is unbiased from the ground truth and its accuracy is better than SRS's, regardless of flow sizes. Also, SRS's variance grows faster as the sampling rate decreases. Fig. 37 shows the CDFs of overall flow-level relative error of both schemes according to the sampling rate. Both were compared with the ground truth. As shown, SketchFlow is more accurate than SRS in all cases where the sampling rates ranged from 0.1 to 0.0001.

Flow Thinning. We evaluated the quality of flow thinning (sampling). Precision refers to the fraction of correctly-sampled flows (*i.e.* sampled flows where the size is equal to or greater than the sampling interval) over all sampled flows. As shown in Table 3, the overall precision of SketchFlow is higher than that of SRS. The precision gap is even greater when the sampling rate decreases. The recall is the fraction of correctly-sampled flows over flows that are supposed to be sampled (*i.e.* all the flows of which sizes are equal to or greater than the sampling interval). As a result, the recall of SketchFlow is shown to be better than SRS in most cases. Overall, the quality of SketchFlow in flow sampling is better than or equal to that of SRS. Note that when the sampling rate is 0.01, the precision is low in comparison with 0.1 and 0.001 due to the drastically-increased mouse flows.

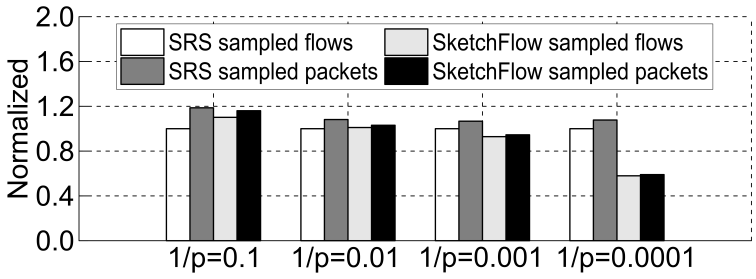
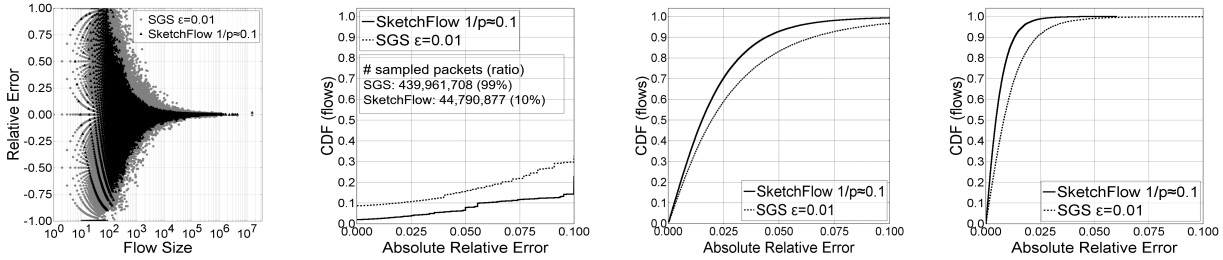


Figure 38: Comparison of mouse flow sampling between SketchFlow and SRS. Mouse flow is a flow which the volume is less than sampling interval p



(a) Relative Error (b) Small Flows (10-1,000) (c) Medium Flows (1,000-10,000) (d) Large Flows (10,000+)

Figure 39: CAIDA trace: Accuracy comparison between SketchFlow and SGS. Both were assigned with 110KB memory for fair comparison. The sampling rate of SketchFlow was 0.1 and the expected relative error of SGS was 0.01. (a) shows the relative error of independent flows. Each point stands for each flow. The closer point to $y = 0$, the better accuracy. (b)-(d) show the CDF of relative error of different flow size intervals.

Packet Thinning. We compared the fraction of the sampled packets over the entire packets of the CAIDA traffic. As shown in Table 4, SketchFlow guarantees the traffic reduction rate, which can relax the overhead under a fixed boundary.

Mouse Flow Sampling. One of the desirable features of SRS is the ability to provide mouse flow samples. The mouse flow is referred to a flow of which the volume is less than the sampling interval p. We note sampling of mouse flows is irrelevant to the size of the flow, which means one-packet sized mouse flows also have a chance to be sampled because of noise in the virtual vector. Fig. 38 shows a comparison between SketchFlow and SRS with respect to the number of sampled flows and the sampled packets. As shown, SketchFlow captures comparable or more mouse flow samples than SRS with sampling rates ($1/p$) of 0.1, 0.01, and 0.001. This illustrates that SketchFlow can be a good alternative to SRS for general-purpose sampling tasks without losing the information of mouse flows, but providing better

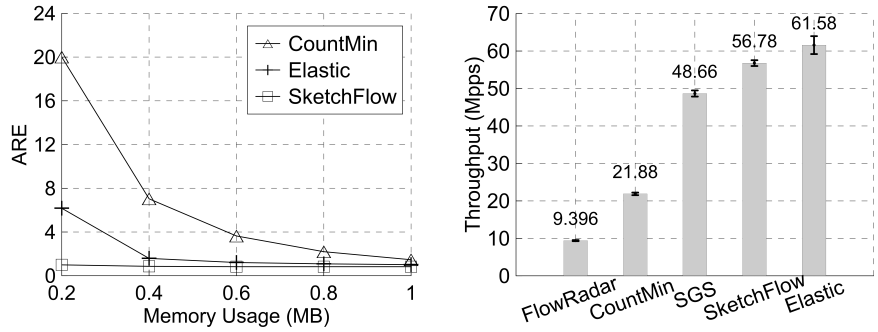
accuracy of elephant flows. Unsurprisingly, though, when the sampling rate is 0.0001, the number of the sampled mouse flows is halved compared to SRS. This is because SketchFlow uses a sketch saturation-based sampling mechanism. Since our dataset follows a heavy-tailed distribution [4], the volume increment of mouse flows following the increment of the sampling interval (p) is slow. Thus, it is hard for mouse flows to saturate the sketch for triggering sampling events. We note that the efficiency of mouse flow sampling of SketchFlow is better than that of SRS with any sampling rate, which means SketchFlow can capture more mouse flows with fewer samples.

4.5.3 SketchFlow vs. Non-linear Sampling Approach (SGS)

We compared SketchFlow with a non-linear scheme, SGS [51]. For fairness, both SketchFlow and SGS used 110KB memory space for their sketch. As shown in Fig. 39(a), the overall relative error of SketchFlow is closer to 0 than SGS's by varying the flow size. Remarkably, SGS outperforms SketchFlow in small flows (Fig. 39(b)) but not large flows (Fig. 39(c)-(d)). The result is reasonable and anticipated because the strategy of SGS is to sample mouse flows with a very high probability, which leads to the frequent sampling of mouse flows. Fig. 39(b) shows that SketchFlow samples only 10% (44M packets), compared to what SGS sampled (440M packets). The estimation of SGS is accurate, and it guarantees the relative error of most flows is within the expected margin ($\epsilon = 0.01$ in our experiments). However, the most critical problem of SGS is packet thinning: in our experiments, SGS triggered 53% sampling events over the entire traffic because a large number of mouse flows appear in the CAIDA trace, the real-world dataset. This unacceptably high sampling rate explains the impracticality of SGS as well as the high accuracy for mouse flows. Unlike SGS, in terms of the flow table overhead (NetFlow) or the network overhead (sFlow), SketchFlow guarantees the desired overhead relaxation rate than SGS.

4.5.4 SketchFlow vs. Sketch Approaches

We further compared SketchFlow with three state-of-the-art sketch approaches: Count-Min [17], Elastic sketch [94] and FlowRadar [55]. We followed experiments in Elastic sketch [94] and divided a one-hour CAIDA dataset into 720 five-second subset traces. We varied the memory usage from 0.2MB to 1MB and evaluated the accuracy in terms of the average relative error ($ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|f_i - \hat{f}_i|}{f_i}$). For Elastic sketch, we fixed the heavy-part with 150KB memory and the remaining for the light-part. For CountMin, we used 3 hash functions as recommended in [31]. In Fig. 40(a), we found that SketchFlow achieves the lowest ARE in all cases, while using similar or even using less memory. On the contrary, accuracy degradation is observed for both CountMin and Elastic sketch following the decrease



(a) Accuracy Comparison (b) Speed Comparison

Figure 40: SketchFlow vs. Sketch Approaches: comparison of memory usage, accuracy and processing speed of sketches on a CPU platform.

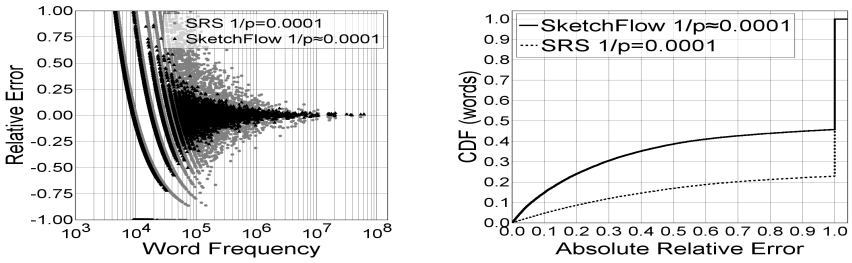
in memory usage. We also conducted the same experiment using FlowRadar, which failed to decode any flow using 1MB memory.

4.5.5 Overhead Evaluation

To evaluate the overheads, we conducted experiments with a testbed that is equipped with Xeon E5-2620 v4 2.10GHz, which supports Streaming SIMD Extensions (SSE).

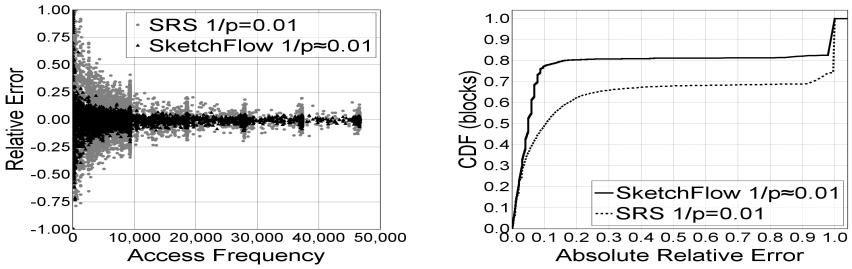
CPU Platform. We evaluated SketchFlow in terms of throughput (Mpps) using a CPU platform. We compared our approach with four solutions (FlowRadar, CountMin, Elastic sketch, and SGS). As shown in Fig. 40(b), SketchFlow achieved higher throughput than FlowRadar, CountMin, and SGS. SGS can reach a throughput of 48.66 Mpps, while SketchFlow is 1.16 times faster (*i.e.* 56.78 Mpps). Remarkably, Elastic achieved the highest throughput (*i.e.* 61.58 Mpps), which is 1.08 times faster than SketchFlow. However, we note that we did not involve any sketch or sample sending in this experiment. Elastic sketch requires a sketch compression process for saving bandwidth overhead caused by sketch delivering.

OpenvSwitch. To comparatively evaluate the overhead of SketchFlow, we integrated SRS (sFlow) and SketchFlow in the packet processing pipeline of OpenvSwitch (using DPDK 17.11.2 [21]). We generated the CAIDA trace using Intel X540AT2 10G NIC and `pktgen` [21] for measuring the average cycles required to make the sampling decision of a packet. In this experiment, a 4-layer SketchFlow was used to approximate the sampling rate of 0.0001 to compare with SRS ($1/p = 0.0001$). According to the experimental results, SRS required fewer cycles (52 cycles/packet), and SketchFlow required slightly more than SRS; 69 cycles per packet. When comparing SRS with SketchFlow, the additional hash computation overhead of



(a) Relative Error (b) CDF

Figure 41: Twitter dataset: Accuracy of SketchFlow and SRS. Both were evaluated with sampling rate 0.0001. Tweet dataset contains ≈7 billion sub-units including word, link, name, etc.



(a) Relative Error (b) CDF

Figure 42: Disk I/O trace: Accuracy of SketchFlow and SRS. Both were evaluated with sampling rate 0.01. Disk I/O trace contains 170 million I/O requests of 390 thousand different offsets.

SketchFlow is large, although can be substantially reduced using hardware-based functions (*i.e.* CRC instruction of SSE), and a few memory accesses are also acceptable for online sampling. Through an in-depth examination, we found that the overhead of SketchFlow occurred mostly in calculating the bit positions of the virtual vector of flows. This overhead can be mitigated by caching the virtual vector of the last flow because a frequent burst behavior of the same flow has been observed in many modern traffic loads [45].

4.5.6 Twitter and Disk I/O trace

We also examined the scalability of SketchFlow using a large dataset (Twitter dataset) and its versatility using a dataset with a different distribution (Disk I/O trace). As results, SketchFlow outperforms SRS for both datasets in terms of the relative error. For the Twitter dataset, we used the sampling rate of 0.0001 by considering its scale. As shown in Fig. 41(b), the overall absolute relative error of SketchFlow is much smaller than SRS.

Moreover, SketchFlow is shown more accurate than SRS for different word frequencies, and the variance of SketchFlow is much smaller (Fig. 41(a)). While the scale of disk I/O trace is much smaller than Twitter's, it presents a different distribution. A sampling rate of 0.01 was used reflecting the fact that most of the blocks were accessed under 10^5 times. As shown in Fig. 42, SketchFlow performs better than SRS in terms of the relative error and variance.

4.6 Related Work

Sampling is implemented using one of two approaches: timer- and packet-driven sampling. Timer-driven sampling is chosen by both sFlow [85] and NetFlow [12, 23]. However, the packet-driven approach is preferred in practice because of its performance. Therefore, several packet-driven approaches have been proposed since its introduction, initially to measure the NSFNET backbone. Claffy *et al.* described three different sampling methods, simple random sampling, stratified sampling, and systematic sampling [14]. Hohn and Veitch [34] compared packet-level sampling's inaccuracy over flow-level sampling's. Duffield *et al.* [22] argued that flow-level sampling is unstable under resource constraints and proposed a threshold-based sampling. In both works, the flow sampling schemes showed higher accuracy than the packet sampling. However, the traffic reduction rate cannot be guaranteed [51].

Another line of works used non-linear sampling rates. Kumar *et al.* [51] introduced a non-linear scheme (SGS) using different probabilities depending on the size of the flows. Their approach acknowledges that information on mouse flows is likely to be lost using a linear approach. SGS employed a compact sketch to record the flows' size with a higher probability for smaller flows. Hu *et al.* [35] and Ramachandran *et al.* [76] introduced similar approaches with different architectures and data structures, providing high accuracy in flow size estimation with mouse flows. However, the high sampling probability of mouse flows leads to a huge number of samples, negatively affecting the traffic reduction rate.

4.7 Conclusion

In this paper, we introduced a new notion of per-flow systematic sampling, where the sampling accuracy is shown to be superior to that of the simple random sampling. To realize this idea, we proposed a new sampling framework using sketches as per-flow samplers. In this framework, a per-flow sketch saturation event works as a signal to sample a packet in a flow, and the per-flow saturation interval as the per-flow sampling interval. Instead of using a sketch as a full flow size estimator that necessarily causes sketch saturation and offline decoding, we had our new sketch algorithm measure only the sampling interval and be emptied for reuse in real-time. With this framework and a sketch algorithm, we successfully

built a highly-accurate sampling algorithm, SketchFlow, which is able to perform per-flow systematic sampling. We showed proof on SketchFlow's accuracy and demonstrated performance by experiment with real-world datasets such as traces from the network, I/O, and social network platforms. We believe that our work opens a new direction in data sampling, and we expect that SketchFlow would inspire more work on per-flow sampling.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[2] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Optimal elephant flow detection. In *Proceedings of the 2017 IEEE International Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9, 2017.

[3] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *Proceedings of the 2017 IEEE International Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9, 2017.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, pages 267–280, 2010.

[5] S. Biswas, J. C. Bicket, E. Wong, R. Musaloiu-E, A. Bhartia, and D. Aguayo. Large-scale measurements of wireless network behavior. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 153–165, 2015.

[6] V. Braverman and R. Ostrovsky. Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, pages 58–70, 2013.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the 1999 IEEE International*

Conference on Computer Communications, INFOCOM 1999, New York, NY, USA, March 21-25, 1999, pages 126–134, 1999.

[8] CAIDA. The cooperative association for internet data analysis, equinix chicago data center. https://www.caida.org. [Apr 06 2016].

[9] CAIDA. The cooperative association for internet data analysis, equinix chicago data center. https://www.caida.org. [13:00-14:00, Apr 19 2018].

[10] M. Chen, S. Chen, and Z. Cai. Counter tree: A scalable counter architecture for per-flow traffic measurement. *IEEE/ACM Trans. Netw.*, 25(2):1249–1262, 2017.

[11] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba. Payless: A low cost network monitoring framework for software defined networks. In *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, pages 1–9, 2014.

[12] Cisco. NetFlow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[13] Cisco. The zettabyte era: Trends and analysis. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html.

[14] K. C. Claffy, G. C. Polyzos, and H. Braun. Application of sampling methodologies to network traffic characterization. In *Proceedings of the ACM SIGCOMM '93 Conference on Communications Architectures, Protocols and Applications, San Francisco, CA, USA, September 13-17, 1993*, pages 194–203, 1993.

[15] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM International Conference on Management of Data, SIGMOD 2003, San Diego, California, USA, June 9-12, 2003*, pages 241–252, 2003.

[16] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in streaming data. *TKDD*, 1(4):2:1–2:48, 2008.

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

[18] Cumulus. Td-routing: Supported route table entries. https://docs.cumulusnetworks.com/display/DOCS/Routing#Routing-SupportedRouteTableEntries.

[19] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. SPHINX: detecting security attacks in software-defined networks. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[20] X. A. Dimitropoulos, P. Hurley, and A. Kind. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *Computer Communication Review*, 38(1):5, 2008.

[21] Data Plane Development Kit. https://www.dpdk.org/.

[22] N. G. Duffield, C. Lund, and M. Thorup. Learn more, sample less: control of volume and variance in network measurement. *IEEE Trans. Information Theory*, 51(5):1756–1775, 2005.

[23] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 30 - September 3, 2004, Portland, Oregon, USA*, pages 245–256, 2004.

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

[25] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[26] S. Gao, Z. Peng, B. Xiao, A. Hu, and K. Ren. Flooddefender: Protecting data and control plane resources under sdn-aimed dos attacks. In *2017 IEEE Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9. IEEE, 2017.

[27] A. A. Ghosh, R. Jana, V. Ramaswami, J. Rowland, and N. K. Shankaranarayanan. Modeling and characterization of large-scale wi-fi traffic in public hot-spots. In *INFOCOM 2011. 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 10-15 April 2011, Shanghai, China*, pages 2921–2929, 2011.

[28] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris. Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments. *Computer Networks*, 62:122–136, 2014.

[29] M. T. Goodrich and M. Mitzenmacher. Invertible bloom lookup tables. In *Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2011, Allerton Park & Retreat Center, Monticello, IL, USA, 28-30 September, 2011*, pages 792–799, 2011.

[30] Google. why is public wifi so slow? https://tinyurl.com/st6vcbp, 2020.

[31] A. Goyal, H. D. III, and G. Cormode. Sketch algorithms for estimating point queries in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 1093–1103, 2012.

[32] A. Hall, O. Bachmann, R. Büssow, S. Ganceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, 2012.

[33] K. Hill. Wi-fi alliance launches 802.11ac wave 2 certification. https://www.rcrwireless.com/20160629/network-infrastructure/wi-fi/wi-fi-alliance-launches-802-11ac-wave-2-certification-tag6, 2020.

[34] N. Hohn and D. Veitch. Inverting sampled traffic. *IEEE/ACM Trans. Netw.*, 14(1):68–80, 2006.

[35] C. Hu, B. Liu, S. Wang, J. Tian, Y. Cheng, and Y. Chen. ANLS: adaptive non-linear sampling method for accurate flow size measurement. *IEEE Trans. Communications*, 60(3):789–798, 2012.

[36] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 113–126, 2017.

[37] IEEE. Ieee 802.11n standard. https://standards.ieee.org/standard/802_11n-2009.html, 2009.

[38] iPerf. https://github.com/esnet/iperf.

[39] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: experience with a globally-deployed software defined wan. In *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*, pages 3–14, 2013.

[40] R. Jang, D. Cho, A. Mohaisen, Y. Noh, and D. Nyang. Two-level network monitoring and management in WLAN using software-defined networking: poster. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2017, Boston, MA, USA, July 18-20, 2017*, pages 279–280, 2017.

[41] R. Jang, D. Cho, Y. Noh, and D. Nyang. Rflow^{+}: An sdn-based WLAN monitoring and management framework. In *Proceedings of the 2017 IEEE International Conference on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9, 2017.

[42] R. Jang, S. Moon, Y. Noh, A. Mohaisen, and D. Nyang. Instameasure: Instant per-flow detection using large in-dram working set of active flows. In *Proceedings of 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 2047–2056. IEEE, 2019.

[43] jFlow. https://www.juniper.net/us/en/local/pdf/app-notes/3500204-en.pdf.

[44] N. Kamiyama and T. Mori. Simple and accurate identification of high-rate flows by packet sampling. In *Proceedings of the 2006 IEEE International Conference on Computer Communications, INFOCOM 2006, 23-29 April 2006, Barcelona, Catalunya, Spain*, 2006.

[45] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *Proceedings of the ACM Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, pages 133–138, 2013.

[46] R. Karp, S. Shenker, and C. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[47] R. Klöti, V. Kotronis, and P. Smith. Openflow: A security analysis. In *2013 21st IEEE International Conference on Network Protocols, ICNP 2013, Göttingen, Germany, October 7-10, 2013*, pages 1–6. IEEE Computer Society, 2013.

[48] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, pages 177–188, 2004.

[49] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, pages 177–188, 2004.

[50] A. Kumar, J. Xu, and J. Wang. Space-code bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications*, 24(12):2327–2339, 2006.

[51] A. Kumar and J. J. Xu. Sketch guided sampling - using on-line estimates of flow size for adaptive data collection. In *Proceedings of the 25th IEEE International Conference on Computer Communication, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2006, 23-29 April 2006, Barcelona, Catalunya, Spain*, 2006.

[52] K. Lan and J. S. Heidemann. A measurement study of correlations of internet flow characteristics. *Computer Networks*, 50(1):46–62, 2006.

[53] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[54] T. Li, S. Chen, and Y. Ling. Fast and compact per-flow traffic measurement through randomized counter sharing. In *Proceedings of the 30th IEEE International Conference on Computer Communications, INFOCOM 2011, 10-15 April 2011, Shanghai, China*, pages 1799–1807, 2011.

[55] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better for data centers. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 311–324, 2016.

[56] P. Lieven and B. Scheuermann. High-speed per-flow traffic measurement with probabilistic multiplicity counting. In *Proceedings of the 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2010, 15-19 March 2010, San Diego, CA, USA*, pages 1253–1261, 2010.

[57] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM Special Interest Group on Data Communication, SIGCOMM 2016, Florianopolis, Brazil, August 22-26, 2016*, pages 101–114, 2016.

[58] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *Proceedings of the 2008 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008*, pages 121–132, 2008.

[59] macof. https://github.com/ggreer/dsniff/blob/master/macof.c.

[60] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner. Openflow: enabling innovation in campus networks. *CCR*, 38(2):69–74, 2008.

[61] Hamming weight. https://software.intel.com/sites/landingpage/ IntrinsicsGuide/#text=_mm_popcnt_u32.

[62] R. Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, 1978.

[63] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 129–143, 2016.

[64] D. Narayanan, A. Donnelly, and A. I. T. Rowstron. Write off-loading: Practical power management for enterprise storage. *TOS*, 4(3):10:1–10:23, 2008.

[65] Netberg. All about bare metal switch. https://bm-switch.com/.

[66] D. Nyang and D. Shin. Recyclable counter with confinement for real-time per-flow measurement. *IEEE/ACM Trans. Netw.*, 24(5):3191–3203, 2016.

[67] OpenDaylight. https://www.opendaylight.org/.

[68] OpenWrt. https://www.openwrt.org/.

[69] OpenWrt. OpenWrt qos-scripts. https://wiki.openwrt.org/doc/uci/qos.

[70] Openwrt. Libpcap ver. 1.5.3-1 ipk for openwrt chaos calmer 15.05. http://archive.openwrt.org/chaos_calmer/15.05/ar71xx/generic/packages/base/libpcap_1.5.3-_ar71xx.ipk, 2020.

[71] Openwrt. Qos (aka network traffic control). https://openwrt.org/docs/guide-user/network/traffic-shaping/packet.scheduler, 2020.

[72] T. A. Pascoal, Y. G. Dantas, I. E. Fonseca, and V. Nigam. Slow TCAM exhaustion ddos attack. In S. D. C. di Vimercati and F. Martinelli, editors, *ICT Systems Security and Privacy Protection - 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings*, volume 502 of *IFIP Advances in Information and Communication Technology*, pages 17–31. Springer, 2017.

[73] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 117–130, 2015.

[74] Y. Qian, W. You, and K. Qian. Openflow flow table overflow attacks and countermeasures. In *European Conference on Networks and Communications, EuCNC 2016, Athens, Greece, June 27-30, 2016*, pages 205–209. IEEE, 2016.

[75] S. Qiao, C. Hu, X. Guan, and J. Zou. Taming the flow table overflow in openflow switch. In M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 591–592. ACM, 2016.

[76] A. Ramachandran, S. Seetharaman, N. Feamster, and V. V. Vazirani. Fast monitoring of traffic subpopulations. In *Proceedings of the 8th ACM SIGCOMM Internet Measurement Conference, IMC 2008, Vouliagmeni, Greece, October 20-22, 2008*, pages 257–270, 2008.

[77] M. Rodrig, C. Reis, R. Mahajan, D. Wetherall, and J. Zahorjan. Measurement-based characterization of 802.11 in a hotspot setting. In *Proceedings of the 2005 ACM SIGCOMM workshop on Experimental approaches to wireless network design and analysis*, pages 5–10. ACM, 2005.

[78] O. Rottenstreich, Y. Kanizo, and I. Keslassy. The variable-increment counting bloom filter. *IEEE/ACM Trans. Netw.*, 22(4):1092–1105, 2014.

[79] Sandvine. Global internet phenomena report. 2016.

[80] S. Sarvotham, R. H. Riedi, and R. G. Baraniuk. Connection-level analysis and modeling of network traffic. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Workshop, IMW 2001, San Francisco, California, USA, November 1-2, 2001*, pages 99–103, 2001.

[81] J. Schulz-Zander, C. Mayer, B. Ciobotaru, S. Schmid, and A. Feldmann. Opensdwn: programmatic control over home and enterprise wifi. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015*, pages 16:1–16:12, 2015.

[82] J. Schulz-Zander, P. L. Suresh, N. Sarrar, A. Feldmann, T. Hühn, and R. Merz. Programmatic orchestration of wifi networks. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 347–358, 2014.

[83] R. T. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM Internet Measurement Conference, IMC 2004, Taormina, Sicily, Italy, October 25-27, 2004*, pages 207–212, 2004.

[84] M. S. Seddiki, M. Shahbaz, S. P. Donovan, S. Grover, M. S. Park, N. Feamster, and Y. Song. Flowqos: Qos for the rest of us. In *Proceedings of the third workshop on Hot topics in software defined networking, HotSDN '14, Chicago, Illinois, USA, August 22, 2014*, pages 207–208, 2014.

[85] sFlow. http://www.sflow.org/.

[86] S. Shin and G. Gu. Attacking software-defined networks: a first feasibility study. In N. Foster and R. Sherwood, editors, *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013*, pages 165–166. ACM, 2013.

[87] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *ACM HotNets*, 2004.

[88] Intel SSE4 Programming Reference. https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf.

[89] W. Sun, O. Lee, Y. Shin, S. Kim, C. Yang, H. Kim, and S. Choi. Wi-fi could be much more. *IEEE Communications Magazine*, 52(11):22–29, 2014.

[90] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. Opentm: Traffic matrix estimator for openflow networks. In *Passive and Active Measurement, 11th International Conference, PAM 2010, Zurich, Switzerland, April 7-9, 2010. Proceedings*, pages 201–210, 2010.

[91] P. Tune and D. Veitch. Towards optimal sampling for flow size estimation. In *Proceedings of the 8th ACM SIGCOMM Internet Measurement Conference, IMC 2008, Vouliagmeni, Greece, October 20-22, 2008*, pages 243–256, 2008.

[92] P. Tune and D. Veitch. Sampling vs sketching: An information theoretic comparison. In *Proceedings of the 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2011, 10-15 April 2011, Shanghai, China*, pages 2105–2113, 2011.

[93] K. Whang, B. T. V. Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, 1990.

[94] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 561–575, 2018.

[95] Y. Yiakoumis, M. Bansal, G. A. Covington, J. van Reijendam, S. Katti, and N. McKeown. Behop: A testbed for dense wifi networks. *Mobile Computing and Communications Review*, 18(3):71–80, 2014.

[96] M. Yoon, T. Li, S. Chen, and J. Peir. Fit a compact spread estimator in small high-speed memory. *IEEE/ACM Trans. Netw.*, 19(5):1253–1264, 2011.

[97] C. Yu, C. Lumezanu, Y. Zhang, V. K. Singh, G. Jiang, and H. V. Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. In *Passive and Active Measurement - 14th International Conference, PAM 2013, Hong Kong, China, March 18-19, 2013. Proceedings*, pages 31–41, 2013.

[98] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 29–42, 2013.

[99] B. Yuan, D. Zou, S. Yu, H. Jin, W. Qiang, and J. Shen. Defending against flow table overloading attack in software-defined networks. *IEEE Trans. Services Computing*, 12(2):231–246, 2019.

[100] M. Zhang, J. Bi, J. Bai, Z. Dong, Y. Li, and Z. Li. Ftguard: A priority-aware strategy against the flow table overflow attack in SDN. In *Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 141–143. ACM, 2017.

[101] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai. Control plane reflection attacks in sdns: New attacks and countermeasures. In M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, volume 11050 of *Lecture Notes in Computer Science*, pages 161–183. Springer, 2018.