Cooperative Network-wide Flow Selection

Ran Ben Basat
Harvard University
ran@seas.harvard.edu

Gil Einziger
Ben Gurion University
gilein@bgu.ac.il

Bilal Tayh
Ben Gurion University
tayh@post.bgu.ac.il

***Abstract*—Network-wide per-flow measurements are instrumental in diverse applications such as identifying attacks, detecting load imbalance, and performing traffic engineering. These measurements utilize scarcely available flow counters that monitor a single flow, but there are often more flows than counters in a single device. Therefore, existing flow-level techniques suggest pooling together the resources of all the network devices. Still, these either make strong assumptions on the traffic or require an excessive number of counters to track all the network flows.**

In this work, we present novel, readily deployable, distributed algorithms that do not require device coordination or assumptions about the traffic. Through an extensive evaluation on real network topologies and network traces, we show that our algorithms attain near-optimal flow coverage in diverse conditions. Specifically, our algorithms reduce the space required to monitor all the flows by up to 4x compared to the best alternative.

I. Introduction

Network-wide measurements are fundamental in applications such as traffic engineering, routing, load balancing, quality of service enforcement, and intrusion detection [35], [23], [31], [34], [43], [46], [56], [58]. In such measurements, *Network Measurement Points (NMPs)* monitor the network and send measurement data to a centralized controller that attains a bird-eye view of the traffic within the entire network [8], [9], [10], [28], [30], [36], [39], [42], [57]. The controller analyzes the collected data for diverse tasks such as identifying superspreaders [40], [51], [56] and elephant flows [13], estimating the flow-size distribution [51], [54], [55] and entropy [25], and detecting micro bursts [27].

Measurement is challenging to implement due to the rapid line rates which force network devices to use fast SRAM memory, which is often too small to store all the flows [41]. Therefore, existing approaches often resort to approximation to save space [15], [16], [26], [29], [14], [17], [45], [52], [19]. Alternatively, per-flow measurement [42], [50], [58] allows accurate monitoring of *a subset* of the flows, which suggests that one can collect network-wide flow-level statistics by partitioning the monitoring task among the network devices. Such *exact* measurements enable important measurement tasks such as identifying elephant flows, hierarchical heavy hitters, and super-spreaders or calculating the flow-size entropy and distribution, without resorting to approximations.

The related work tries to encode as many flow entries as possible in the high-performance SRAM memory [38]; however, some network devices may still lack the space to monitor all the flows that traverse through them. Other works pursue some form of cooperation between the NMPs. For example, in cSamp [50], the controller calculates an efficient assignment of flows to NMPs to maximize the number of monitored flows subject to the capacity limitations of each device. However, cSamp's controller requires the routing of all the flows to perform this calculation. Alternatively, *Flow-Radar* [42] shares the limited flow-counters between multiple flows, and use an offline *network decode* that processes the flow-counters to maximize the number of decoded flows. Our work avoids shared flow counters as they require decoding the flows from all network devices. Instead, we treat the monitoring task as a distributed covering problem where each NMP independently selects a subset of the flows to monitor. The goal is then to maximize the number of monitored flows network-wide, i.e., those that are tracked by at least one NMP.

978-1-7281-6992-7/20/$31.00 ©2020 IEEE

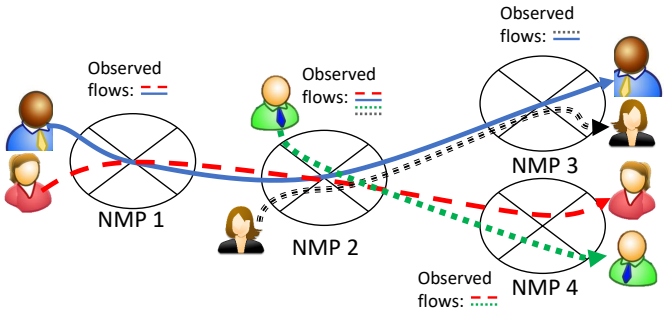


Fig. 1: An example of flow-based monitoring, three network flows traverse a network of four NMPs. An optimal solution can monitor all flows even if each NMP can track a single flow, for example, by monitoring the blue (solid) flow in NMP 1, the green (dotted) flow in NMP 2, the black (compound) flow in NMP 3, the red (dashed) flow in NMP 4. Our goal is to minimize the overlap between the flow-sets that are tracked by different NMPs, and thereby maximizing the overall number of monitored flows, without switch to switch communication.

A. Our Contribution

We introduce the *Cooperative Flow selection (CFS)* algorithm where NMPs independently select which flows to monitor without traffic maps, and without explicit coordination. CFS leverages the Time To Live (TTL) field of the IP header to reduce the overlap between the flow sets tracked by different NMPs; thereby, we maximize the number of flows that are monitored at least at one NMP. Intuitively, as the TTL value is reduced by one for each routing hop, different NMPs that observe the same packet, see it with a different TTL value. The problem is as illustrated in Figure 1.

We evaluate CFS on four real network data, and two network graphs and show that it attains a high (e.g., 95%) flow coverage using a small amount of memory. Despite its effectiveness with a small number of counters, reaching perfect coverage using CFS requires a large amount of space. Therefore, we suggest the CFS-FR algorithm that efficiently integrates CFS with the previously proposed Flow Radar [42] algorithm to attain perfect coverage (1.0). Our evaluation shows that CFS-FR requires less memory than CFS to achieve complete flow coverage and that both CFS and CFS-FR offer a better space/coverage tradeoff compared to Flow-Radar. Our algorithms can also estimate the total number of flows in the measurement, which increases the accuracy of identifying superspreaders and of estimating the flow's size distribution. This capability makes our algorithms useful for a wide range of memory configurations where other algorithms fail. In addition, our algorithms can detect routing loops in real time, and without making assumptions about the routing. In comparison, Flow-Radar only detects such loops at the end of the measurement, and while assuming that flows are always routed on shortest paths. Our experiments also include applications such as identifying heavy hitters and superspreaders, estimating the flow-size distribution, and detecting routing loops. In all tasks, our algorithms are more accurate than the alternatives.

II. Preliminaries

This section provides the necessary background and explains the model and measurement tasks that we address.

A. Background

Network measurement solutions target software [44], [16], [9], hardware [41], [52], [48], [42], or a combination of the two [35]. The rapid line rate and traffic volume make measurement challenging in all deployments, but the typical bottlenecks vary between hardware and software. In hardware, real-time processing of packets at the line's rate requires the usage of scarcely available SRAM memory. Further, hardware switches have restricted memory access patterns and often cannot perform even arithmetic operations such as multiplication [48]. Software processing offers significant flexibility. Despite this flexibility, software implementations are often limited by processing power. That is, we cannot perform complex processing for every packet as it results in excessive CPU usage [44]. Therefore, software solutions often use some form of packet sampling to reduce the processing overheads [16], [44]. While some measurement algorithms target a single device [52], and are useful for local optimizations, the current trend is to offer network-wide solutions that provide a bird-eye view of the entire system [36]. Such measurement is essential for numerous SDN optimizations such as traffic engineering, detecting load imbalance, and for finding traffic anomalies such as port scans [40], and superspreaders [56].

Measurements can also be packet- or flow-based, and the vast majority of measurements are packet-based [20], [48]. Flow-based measurements [50], [37], [32] focus on exact measurement a subset of the network flows. Such measurements often use hardware capabilities of per-flow counters [6]. Such counters can monitor a flow and track its number of packets or byte-size. Next, we survey leading flow measurement techniques to position our work.

1) cSamp: cSamp [50] treats flow sampling as an optimization problem. It receives routing paths of all flows and the capacity constraints (number of flow counters) of every NMP and searches for an optimal assignment of flows to NMPs. Such an assignment maximizes the total number of monitored flows without violating the capacity constraints for NMPs. In practice, the flows are not always known prior to the measurement, which implies that cSamp requires an additional measurement mechanism to identify new flows and their paths. Also, when new flows appear, cSamp may radically change the assignment of flows to NMPs. Indeed, our algorithms can support cSamp, which in turn will guarantee that our assignment is optimal. However, our evaluation shows that the benefit of such coordination may be limited since our algorithms already achieve near-optimal flow coverage in diverse settings.

2) Flow-Radar: Flow-Radar [42] (FR) shows that flow-based measurement is feasible even within the restricted programming model of programmable switches. FR maintains a Bloom filter [24] (BF) on each NMP to approximate the set of flows that passes through it. FR also maintains multiple (three, in the suggested configuration) arrays of flow monitoring entries (D). Each entry contains (i) a flow identifier (id), (ii) packet and byte counts (pc, bc), and (iii) a flow count (fc).

Once a packet arrives, Flow-Radar first tests its flow identifier for membership in the Bloom filter. If the flow identifier is not a member, then it is added to the filter, and we know that we encountered a new flow. In that case, FR uses multiple hash functions (h_i for array i), to select a single entry in each array. Multiple flows may collide on the same entry, in which case we bitwise-xor their identifiers into the flow id field. That is we xor the newly arriving flow identifier (x) with $D[h_i(x)].id$, and increase the flow count ($D[h_i(x)].fc$) by one. Next, we update the packet and byte count fields according to the arriving packet. This process is applied to each array.

The decoding process leverages on the reversibility of the xor operation (we can undo an xor operation by performing it again). At the end of the measurement, the controller collects all entry arrays form all the NMPs and begins a *Network decode* process. In that process, the controller seeks an entry that only contains a single flow, and records the flow identifier along with its packet and byte counts. If the controller finds such entry for flow (x), it removes x from the measurement by going over the arrays of all NMPs in its path and remove x from their encoding. That is, FR xors x's identifier in the $D[h_i(x)].id$ field, reduces $D[h_i(x)].fc$ by one and decreases $D[h_i(x)].pc$, and $D[h_i(x)].bc$ by the $x's$ packet- and byte-count respectively. By removing x, FR can potentially find a new counter with a single flow ($D[h_i(x)].fc = 1$), and the process repeats until convergence (that is, until there are no decodable flows).

Algorithm 1 provides pseudocode for the packet handling of each NMP in FR. Notice that in Line 2 we test if x is part of the monitored set (using the filter BF). If the flow is not part of the set, we treat it as a new flow and add it to the filter (Line 3). We also xor the flow identifier (x) in a single entry of each array and increment the flow count of that entry by 1 (Line 5 and Line 6). After Line 6, we know that the flow is already within FR's arrays; We increase the packet count of corresponding entries by one and their byte count by the packet's size (w). Note that parameter t represents the packet's TTL, which is not used by FlowRadar. We keep it as part of the packet's description so that all pseudocode retains the same style.

Algorithm 1 The Flow-Radar Algorithm

```
1: procedure HANDLE PACKET(⟨x, t, w⟩)
2:     if x ∉ BF then
3:         BF.add(x)
4:         for i = 0; i < nrArrays; i = i + 1 do
5:             D[h_i(x)].id⊕ = x                    ▷ bitwise-xor
6:             d.fc = d.fc + 1
7:     for i = 0; i < nrArrays; i = i + 1 do
8:         D[h_i(x)].pc+ = 1
9:         D[h_i(x)].bc+ = w
```

B. Model and Notations

The network data consists of a *stream* of packets $\mathcal{S} \in (\mathcal{U} \times \mathbb{N} \times \mathbb{N})^*$, where each packet $\langle x, t, w\rangle$ is associated with a *flow identifier* $x \in \mathcal{U}$, a time to live (TTL) value ($t \in \mathbb{N}$), and a weight ($w \in \mathbb{N}$). Flow identifiers refer to 5-tuples that include the source and destination IP addresses and ports as well as the protocol field. In general, we can derive flow identifiers from various packet header fields,

The network consists of z *Network Measurement Points (NMPs)* $R_1, \ldots R_z$, and the routing for all packets from a particular flow follows a sequence (an ordered set) of NMPs. The notation $\mathcal{O}(x)$ denotes the ordered list of NMPs that corresponds to the routing path of flow x. The TTL value of a packet is reduced by one at each NMP. We assume that the initial TTL value is the same for packets and starts from 255. The TTL IP header is an 8-bit field, which means that its value is between 0 and 255. We note that this assumption seems standard, for example, in data center scenarios where the operator has complete control [21], [49]. In an ISP setting, we may be able to set the TTL value at the ingress switch. Finally, if we wish to avoid using the TTL, we can add a small counter to packets at the first hop. each NMP observes all the packets that pass through it with the appropriate TTL values. Formally, NMP i sees a substream $\mathcal{S}_i \subseteq \mathcal{S}$ such that $\cup_{i=1}^{z} \mathcal{S}_i = \mathcal{S}$ (i.e., each packet passes through at least one NMP). Our model is similar to that of [42], [13].

The *size* of a flow $x \in \mathcal{U}$ is its number of packets, or the sum of byte-size of its packets, depending on the context.

Our model has a single controller that collects the data of all NMPs at the end of the measurement. For ease of reference, the notations used in this work are summarized in Table I.

Symbol	Meaning
$\mathcal{S}$	The packet stream
NMP	Network Measurement Point
z	Number of NMPs
S_i	Traffic arriving at NMP i.
$\langle x, t\rangle$	A packet from flow x with TTL t
$\mathcal{O}(x)$	The routing path for flow x.
$\mathcal{U}$	The universe of flow identifiers
f_x	The frequency of flow $x \in \mathcal{U}$
θ	Heavy hitter threshold
k	Fat-tree parameter.
χ	Number of entries per NMP.
ψ	Superspreader threshold.
$T(t)$	Target function, T maps a TTL value (t) into a point on the unit circle $(0, 1)$.
$h(x)$	Hash value of flow x, h maps a flow identifier (x) into a point on the unit circle $(0, 1)$.
α	Portion of memory allocated to CFS in the CFS-FR algorithm.

TABLE I: A list of symbols and notations

C. Measurement Tasks

Here, we describe the measurement task that we address.

1) per-flow monitoring: The goal of per-flow monitoring is to monitor as many flows as possible accurately. That is, each NMP can monitor a fixed number of flows, and NMP R_i can monitor flow x if $R_i \in \mathcal{O}(x)$. That is, an NMP can only monitor a subset of the flows that it routes. When an NMP monitors a flow, it maintains a counter that measures that flow's size. It updates the counter for each arrival of the flow's packets.

The *flow coverage* is the portion of flows that are monitored by at least one NMP. A flow coverage of 1.0 means tracking all flows, while a flow coverage of 0.8 refers to monitor 80%.

2) Flow Size Distribution: Our goal here is to estimate the distribution of flow sizes across the network. That is, we estimate the number of flows with each possible size. We measure the quality of the flow size distribution estimation using *Root Mean Square Error*, comparing each estimated frequency to the actual number of flows with that size.

3) Super Spreaders: A *Superspreader* is a source IP address that communicates with more than ψ destination IP addresses. Such addresses are the origin of many flows and are thus visible in flow-based measurements. We measure the quality of a superspreader set by the $F1$ score. The $F1$ score factors both *false positives* (wrongly identified superspreaders), and *false negatives* (missed superspreaders), as follows: $2 \cdot \frac{(1-FPR)\cdot(1-FNR)}{(1-FPR)+(1-FNR)}$ where FPR is the false positive ratio, and FNR is the false negative ratio.

4) Heavy Hitters: Given a threshold θ, the *heavy hitters* (also known as elephant flows) are all the flows whose size is larger than θ. We use the F1 score to determine the quality of a set of heavy hitters (in the same manner as with superspreaders).

III. Cooperative Flow Selection

We now introduce our algorithm, named *Cooperative Flow Selection (CFS)*. The intuition behind CFS is to view the network-wide flow-based measurement as a covering problem where each NMP selects a subset of the flows (that traverse through it) to monitor. Our goal is to maximize the total number of monitored flows among all the NMPs. For simplicity, CFS follows a straightforward design pattern where the NMPs reach independent decisions and do not interact with other network entities. Such a design pattern reduces the complexity of implementing CFS and makes it easier to deploy in practice.

A. NMP Structure and Operation

Our NMPs can monitor up to χ flows using their χ monitoring entries. NMPs associate flows *grades* that depend on their IDs and the TTL value. Each entry contains a flow identifier, its packet and byte counts, and its grade. Once a packet from a monitored flow arrives, we increase its packet count by one and its byte count by the packet's size. Our work seeks the *exact* packet and byte count for monitored flows. Therefore, each NMP must decide if to monitor a flow once encountering its first packet. We achieve this by using a pseudo-random but deterministic rule that assigns a grade for each flow. The calculation of such scoring is deterministic, which means that all packets of the same flow receive the same grade. Each NMP monitors the χ minimal-grade flows with the that it encounters. That is, if an NMP already tracks χ flows and see a new flow x, it will only admit (start monitoring) x if its grade is smaller than one of the currently tracked flows. In this case, the NMP stops tracking the previous flow and allocates the entry for x (setting the packet count to one). Note that at each point in time, the set of monitored flows received a counter at their first packet, and thus the packet and byte counts are exact.

B. Determining the Grade

We achieve implicit cooperation by having NMPs assign different grades to each flow. Specifically, we strive that flows will only be monitored at one NMP when space is tight, and thus we aim that a flow will not get low grades from multiple NMPs. Intuitively, one can assign a pseudo-random uniform grade at each NMP, but this solution still enables "lucky" flows to receive low grades in multiple NMPs. Instead, we leverage the TTL field to make sure that the grades are *anti-correlated* to each other.

We assign flow (x) with a hash value ($h(x)$), uniformly distributed in $(0, 1)$. We also assign each TTL value (t) with a *target value* ($T(t)$). The grade of each flow is defined as the distance between the flow's hash and the target value. That is:

$$grade(\langle x, t\rangle) = |h(x) - T(t)|.$$

The strategy for determining $T(t)$ follows in Section III-C.

As target values are points on $(0, 1)$, $h(x)$ is compared to a different point on each hop, and it cannot be simultaneously very close to multiple different points. For example, if $h(x)$ is close to 0.5, then it is not too close to 0.2. Thus, NMPs grade the same traffic differently and prioritize different flows.

C. Determining Target Values

Ideally, we would like the target values encountered by each flow to be evenly spread along $(0, 1)$ as it would maximize the diversity of monitored flows. However, it is impossible to distribute the target values evenly as the same target value may apply to flows with different routing path lengths.

We suggest two approaches for determining the target values. The *Greedy* approach attempts to maximize the minimal distance between the target values for an unknown path length, while *Folding* implicitly assigns a large number of target values for each TTL to evenly spread the points for each flow.

1) The Greedy Approach: We start by setting the first target value at $T(255) = 0.5$. Then we determine $T(254)$ to be the point with maximal distance from $T(255)$, e.g., $T(254) = 0.25$ or 0.75. We continue with T Similarly, $T(253)$ is a point that maximizes the minimal distance to $T(255)$ (e.g., 0.75).

Simply put, this selects the next target to be farthest from previous targets. The greedy strategy is not optimal but it is simple to implement, and works reasonably well empirically. That is, we can pre-compute the target values, and store them on a small hash table.

2) The Folding Approach: Since the Greedy approach does not distribute the target values uniformly for most routing lengths, we suggest the Folding approach which behaves better for varying routing path lengths.

Determining Target values: For a hop number y, we determine the target values to be: $T(254 - y) = 2^{-(y+1)}$, that is, 0.5 for $TTL = 254$, 0.25 for $TTL = 253$, and so forth.

In the Folding approach, the target values are not compared to the flows hash value ($h(x)$), but to modified hash values $\bar{h}(x, t)$ for flow x, and current TTL t. These values are computed as we explain below.

Computing modified hash values: Given $h(x)$, and denoting $y = 254 - t$, we define $\bar{h}(x, t) \triangleq h(x) \mod 2^{-y}$. For example, $\bar{h}(x, 254) = h(x)$, $\bar{h}(x, 253) = \begin{cases} h(x) & \text{if } h(x) < 0.5, \\ 0.5 - h(x) & \text{otherwise.} \end{cases}$.

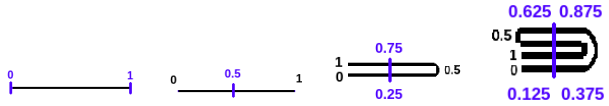
Scaling the grade: Since $T(254 - y) = 2^{-(y+1)}$ and $\bar{h}(x, 254 - y)$ is in the range $[0, 2^{-y}]$, we need to scale the distance to enable comparison of flows that visit the NMP with different TTL values. Therefore, we define the grade as $grade(\langle x, t\rangle) \triangleq \begin{cases} min(1 - h(x), h(x)) & \text{if TTL=255,} \\ 2^{254-t} \cdot \left|\bar{h}(x, t) - T(t)\right| & \text{otherwise.} \end{cases}$

Intuition: We would like to allow each additional NMP down the path of a flow to have a greater chance of tracking it, *if it is not yet monitored*. Using the Folding approach, the first hop tracks flows for which $h(x) \approx 0$ *or* $h(x) \approx 1$. The next hop aims to track those that are not tracked yet, so it "folds" the range to consider flows for which $h(x) \approx 0.5$. Similarly, the third hop should monitor flows for which $h(x) \approx 0.25$ *or* $h(x) \approx 0.75$ and the fourth hop monitor flows for which $h(x) \approx 0.125, 0.375, 0.625, 0.875$. By folding the range once more we get that flows for which h is furthest from the values that the previous hops tracked are more likely to be tracked. Figure 2 illustrates this approach.

D. Estimating the flow coverage

Estimating the obtained flow coverage is useful to analyze the collected data better. For example, superspreaders are source IP addresses that communicate with more than ψ destination IP addresses. If the flow coverage is 0.2, it is best to report source IPs that communicate with 0.2ψ destination IPs in the collected data. Thus, estimating the flow coverage increases our accuracy in some measurement tasks.

The controller knows how many different flows were selected, but not many flows appeared in the measurement. We



(a) first hop (b) second hop (c) third hop (d) fourth hop

Fig. 2: A graphical explanation of the Folding approach. For TTL = 255 the flows that are closest to 0 and 1 are selected. In the next hop, we select the flows with hash that's closest to the middle (0.5), and then (TTL = 253) we fold one time and track the flows closest to the middle (0.25 and 0.75), similarly in the fourth hop we fold two times and we track those in the middle.

suggest estimating the total number of flows in the measurement by borrowing the technique from [13]. The technique utilizes mergable count distinct (CD) algorithms (e.g. [33], [12]) to estimate the number of distinct packets in the network.

CD algorithms monitor some observable (e.g., the minimum hash value) of all the monitored flows. If the minimum value is 0.01, then we estimate that there are 100 different flows. We use multiple such estimators to improve accuracy. The work of [13] suggests to use a mergeable count distinct instance in each NMP, and merge these at the controller to estimate the total number of packets within the measurement. Thus, we trivially extend the method to estimate the total number of flows. Count distinct algorithms support three methods: (i) add that adds a new flow, (ii) $query$ that returns an estimation of the number of *distinct* flows added, and (iii) $merge$ that merges two instances of the algorithms into an instance that estimates the total number of distinct items in both algorithms.

E. Handling Single Hop Routing Paths

Consider flows that only traverse a single NMP. Such flows are the least likely to be selected by CFS as they only get one chance. However, selecting these flows never decreases the total number of monitored flows as there is only one NMP that can monitor them. Therefore, we always monitor single NMP flows at the expense of other flows. NMPs identify as their destination IP address is the same as the next hop.

F. The CFS algorithm

Algorithm 2 presents a pseudocode of CFS. Here, Line 2 updates a count distinct algorithm with the flow identifier. Line 3 handles monitored flows and increments their packet and byte counters. The notation D is a collection of entries, where each entry contains the flow identifier, grade, packet counter, and byte counter. Line 7 calculates the hash value of the packet, following with the grade calculation in Line 8. In this pseudocode, we always add the packet to D, but then in Line 10, we evict an entry if the size of D becomes larger than χ, that is at the end of HandlePacket, the size of D is at most χ. The method Evict (in Line 11) evicts the minimally graded item. To implement the short routing paths optimization, we design the evict method to avoid evicting flows that have a single-hop path. For simplicity, we avoid the additional notation to describe the evict method formally. However, our CFS implementation includes this optimization.

Algorithm 2 Cooperative Flow Selection (CFS)

```
1:  procedure HANDLE PACKET(⟨x, t, w⟩)
2:      CDAlg.add(x).
3:      if x ∈ D then
4:          D[x].packets+ = 1
5:          D[x].size+ = w
6:          return
7:      h_x = h(x)
8:      grade = Distance(h(x), T(t))
9:      D.add(Entry(x, grade, 1, w))
10:     if D.size() > χ then
11:         D.Evict()
```

G. Limitations of CFS and the Synergy with Flow-Radar

Through experimentation, we discover that CFS is almost optimal until we reach high flow coverage values (e.g., 0.95). However, closing the gap between 0.95 and 1.0 is inefficient (in some cases, it requires doubling the per-NMP space). Since it is costly to double (or more) the memory only to capture the last 5% flows, CFS may become ineffective when targeting full coverage. Intuitively, while we can increase the per NMP memory until 100% of the flows are monitored, doing so is ineffective since there is a small number of flows that receive relatively bad grades from all the NMPs in their path. Thus, if we are to monitor these flows, we also need to monitor all the flows that received better grades.

Surprisingly, Flow-Radar's ability to remove already identified flows from the measurement allows us to do just that. That is, we can run Flow-Radar in parallel to CFS, and then remove the flows that were monitored by CFS from Flow-Radar. That is, we get to run Flow-Radar only on the flows that CFS fails to capture. That is, if a flow is not tracked by CFS we add it to Flow-Radar. Similarly if a the current flow replaces a higher-grade flow, we add the kicked flow to Flow-Radar with its current packet and byte counts.

Before we decode flows from Flow-Radar, we remove the CFS flows from all the Flow-Radar instances similarly to Flow-Radar's network decode process. That is, we do the following: (i) xor their identifier from the identifier field, (ii) reduce the number of xor-ed flows by 1, and (iii) update the packet and byte counts to remove the packet and byte count of the CFS flow. Doing that brings us to a point where we have Flow-Radar instances that monitor *only* the flows CFS misses. We proceed with these instances normally and decode the missed flows according to Flow-Radar's network decode algorithm.

Algorithm 3 provides pseudo-code for the unified CFS+Flow-Radar (CFS-FR) algorithm. As shown, we use both algorithms in parallel, and the good synergy happens during the controller's decode phase. Intuitively, when using CFS-FR, we need to decide how much memory to allocate to CFS, and how much memory to allocate to Flow-Radar. The parameter $\alpha \in (0, 1)$ determines the ratio between the two algorithms. Specifically if CFS-FR is allocated χ entries, we allocate $\alpha \cdot \chi$ entries to CFS, and $(1 - \alpha) \cdot \chi$ entries to Flow-Radar.

1) Optimization: Although Algorithm 3 is simple, it can be slightly optimized. Intuitively, the Bloom filter of the Flow-

Algorithm 3 Unified CFS + Flow-Radar (CFS-FR) Algorithm

1: **procedure** HANDLE PACKET($\langle x, t, w \rangle$)
2: $CFS.HandlePacket(\langle x, t, w \rangle)$
3: **if** a flow is not tracked by CFS **then** Add it to FR

Radar instances needs to handle all the flows in the measurement even when we allocate Flow-Radar with a small number of entries. A simple optimization is to break the black-box abstraction and implement a method to move flows between CFS and Flow-Radar with multiple packets. That method differs from the handle packet method as it increments the packet and byte counts by the entry's value. We start by monitoring all flows with CFS and do not update Flow-Radar. However, when CFS evicts a flow (see line 11, in Algorithm 2), we insert that flow to Flow-Radar that continues to monitor future packets of that flow. That is, our modified Flow-Radar instance only monitors flows that we moved to it. Thus, we only need the Bloom filter for the flows observed by Flow-Radar, which enables us to use a smaller Bloom filter for Flow-Radar.

H. Implementing CFS in practice

The CFS and CFS-FR algorithms are simple and can be implemented in software switches such as OVS with minimal overhead. Specifically, we can use the q-MAX algorithm [18] to track the χ flows with the smallest grade with $O(1)$ worst-case update time per-packet. Our algorithms are also readily deployable on PISA programmable switches using suitable modifications. First, we note that almost all operations (including the Folding approach) can be implemented using simple supported operations such as hashing, addition, and bitwise-shift and bitwise-and. The main challenge for such deployment is in finding the minimally graded flow. Indeed, finding the minimum in a large counter-set is a problem in PISA switches [48], [52]. We overcome this difficulty as follows. We store the counters in a $d \times w$ matrix such that each column is stored in the memory of a pipeline stage. Each flow is hashed into one of the d rows, and each row tracks the w flows with the minimal grade mapped to it using a rolling maximum (see [52]). That is, if the current flow has a higher grade than another in its row, we replace the flow, adding the previously stored one to the packet header vector, which would trigger a sequence of displacements to maintain the structure. Leveraging the Balls-and-Bins framework, we can derive bounds on d and w to get that the χ flows with the minimal grade will be stored except with a user-defined probability. That is, for any desired success probability (e.g., 99.9%), we can set d, w such that no more than w flows among the χ with minimal grade are mapped into any specific row. Therefore, the hardware deployment requires slightly more than χ counters for the same performance, and the actual number depends on the required success probability.

The following theorem, which follows from the analysis of [53, Theorem 7], assumes that the algorithm is allocated with d rows (corresponding to per-stage memory) and bounds the required w (that corresponds with the number of pipeline stages) required for tracking *all of the χ flows* (i.e., it will be identical to the software implementation). For example, if we have $\chi = 10000$ with probability 90% (and thus, $\delta = 0.1$) and have $d = 1010$ then we use $w = 27$. Having more space (larger d) reduces w; e.g., with $d = 10000$ we require just $w = 11$. Further, we note that a failure to capture a flow at the switch does not mean that it will not be tracked as it may be monitored by a different switch, and that the additional space used on the switch allows tracking additional flows and potentially increasing the coverage further.

Theorem 1. *Let $d \in \mathbb{N}^+$, let $\delta > 0$ be a user-defined error probability and let*

$$w = \begin{cases} e \cdot \chi/d & \text{if } \chi > d\ln(2d/\delta) \\ e \cdot \ln(2d/\delta) & \text{if } d \cdot \ln \delta^{-1}/e \le \chi \le d\ln(2d/\delta) \\ \frac{1.3\ln(2d/\delta)}{\ln\left(\frac{d}{\chi \cdot e}\ln(2d/\delta)\right)} & \text{otherwise} \end{cases}.$$

Then no more than of the χ top-score flows are mapped into the same row, and thus the algorithm tracks all of them, with probability at least $1 - \delta$.

I. Computing the Optimal Flow Coverage

In our experiments, we would like to quantify the gap between our algorithms and the optimal attainable coverage. We assume knowledge of the routing of all flows, and the possibility to set the monitored flows by each switch. We optimize the flow coverage under the capacity constraint of the switches.

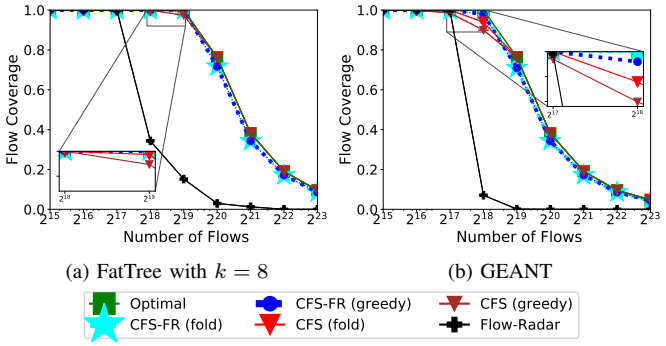
To that end, we model the network as a *directed* graph $G = (V, E)$ (such that each link is modeled as two directed edges) and consider functions $c : E \to \mathbb{N}$, $M : V \to \mathbb{N}$, such that $c(e)$ denotes the number of flows whose paths go through e and $M(v)$ denotes the number of flows a switch v can track (its memory). We further denote by $S(v)$ the number of flows that *originate* from v (i.e., that are connected to v). We build a *flow network* $N = (G', \bar{c}, s, t)$, where $G' = (V \cup \{s, t\}, E \cup \{(s, v) \mid S(v) > 0\} \cup \{(v, t) \mid M(v) > 0\})$, and $\forall (s, v) : \bar{c}((s, v)) = S(v), \forall (v, t) : \bar{c}((v, t)) = M(v)$ and $\forall e \in E : \bar{c}(e) = c(e)$. That is, we add two artificial nodes s, t, connect s to all switches that have end-hosts, and connect all stateful switches to t. The capacity of an arc (s, v) is the number of flows that originate from s and the capacity of (v, t) is the memory of switch, $M(v)$.

Using the above network, we can run a Max Flow algorithm such as Edmonds-Karp or the state of the art algorithm by Orlin [47] to efficiently compute the number of flows that the optimal solution can track. Specifically, the flow value $f((v, t))$ tells us how many flows the switch v monitors and $f((s, v))$ is the number of monitored flows whose source is s.

J. Detecting Packet Loss and 'Black Holes'

When the same flow is monitored in multiple NMPs (e.g., when there is enough per-NMP space), a possible difference between their counts is either due to packet loss or to in-flight packets that passed a one NMP but did not reach the other yet. Distinguishing between transient packets and actual loss is possible once the flow terminates, or when we know the characteristics of the network (e.g., its maximal delay).

In networking, black holes refer to places in the network where incoming or outgoing traffic is silently discarded due



(a) FatTree with $k = 8$ (b) GEANT

Fig. 3: Effect of the number of flows on the flow coverage for fixed per-switch memory of 10,000 flow entries.

to some fault or a misconfiguration. By periodically collecting the traffic from all NMPs, the controller can infer the existence of a black hole by finding a flow that is monitored by two different NMPs but only increased its packet count in one of them. Another indicator for short-lived black holes is a large number of lost packets on a specific flow that drastically exceeds the normal loss.

K. Detecting Routing Loops

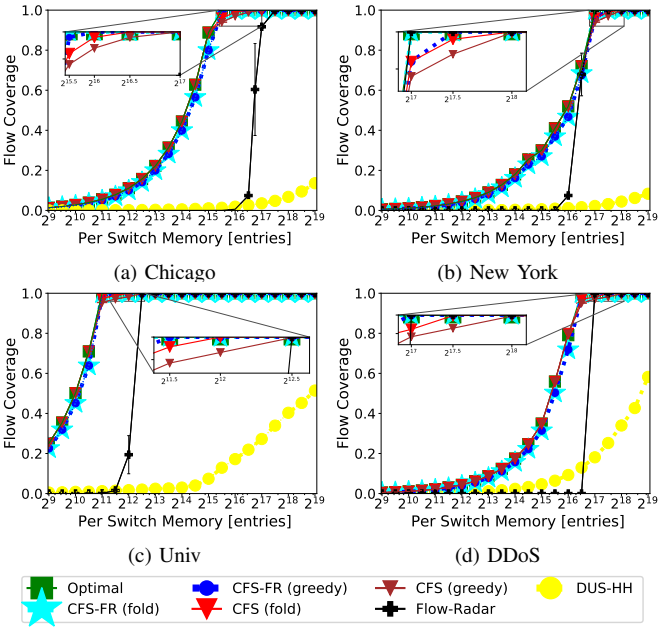
We extend CFS to detect suspected routing loops. Intuitively, in the case of a routing loop, the same NMP would get the packets of a monitored flow multiple times with different TTL values. Thus, to detect this case, we assign an additional byte to each flow monitoring entry, which documents the TTL of the monitored flow. CFS suspects a routing loop when encountering packets of a monitored flow with a different TTL field. In that case, it can notify the controller of a possible routing loop. Note that this is not a definite indication as routing changes can also cause such an event. However, when we encounter a routing change, then the report of suspect routing loops would cease once the system stabilizes. Thus, we only conclude that a loop exists if we observe several distinct TTL values for the same flow within a short timeframe.

IV. Evaluation

In this section, we evaluate CFS and CFS-FR and explore their benefits and limitations compared to prior works. Section IV-A explains the methodology of our evaluation, while Section IV-B contains the evaluation itself.

A. Methodology

1) Evaluated Algorithms: We compare our CFS, and CFS-FR algorithms (denoted CFS, and CFS-FR respectively) with Flow-Radar [42] (denoted Flow-Radar), and the recent heavy hitter identification algorithm [13] (DUS-HH). The notations CFS (fold) and CFS (greedy) denote the CFS algorithm with the fold and greedy approaches respectively. We configure the CFS-FR algorithm with $\alpha = 0.9$, which means that we allocate 90% of the entries to CFS, and 10% to Flow-Radar. We configure Flow-Radar with the authors suggested configuration of three hash functions. We compare our algorithm with Flow-Radar for a given number of *per-switch entries*. Each entry in our algorithm has a flow ID, grade, packet count and byte count. In Flow-Radar, instead of grade, entries



(a) Chicago (b) New York

(c) Univ (d) DDoS

Fig. 4: Flow coverage when varying the per switch memory for the Fat Tree network topology (K=8).

have flow-count. Therefore, we estimate that each entry in Flow-Radar has a similar byte-size to ours, making this a fair comparison. Flow-Radar also requires additional memory for its Bloom filters. In our implementation, we replace the filters with (false-positive free) lists and do not consider this space in the comparison. We compare the algorithms with the same number of per-switch entries. The actual memory required for their representation is similar for equal number of entries.

We denote by Optimal, the optimal attainable flow coverage as calculated by our algorithm in Section III-I.

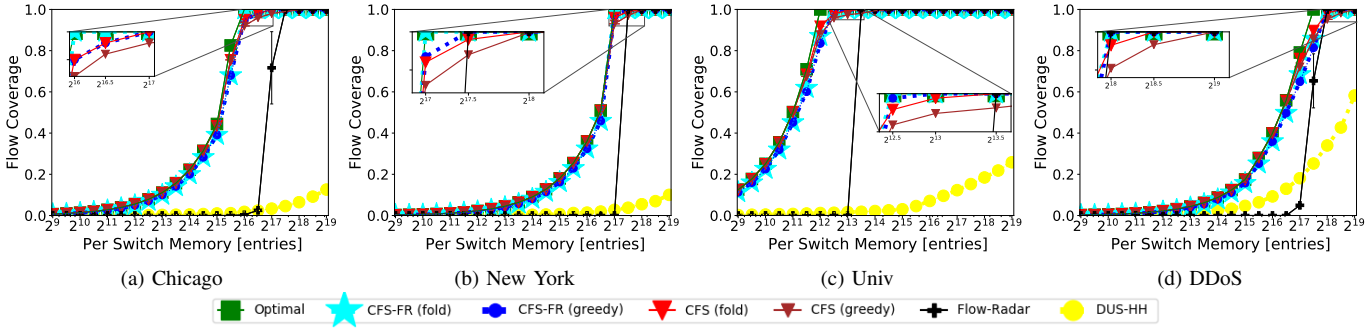
2) Network Topology: We evaluate our algorithms on two different networks; a Fat Tree topology, which is common in data centers, and the GEANT pan-European research and education network [7]. The GEANT network consists of 40 switches, ordered in a mesh without an apparent hierarchical structure. The Fat Tree topology uses parameter $k = 8$ [5] and consists of 80 switches, and 128 hosts. In our evaluation, we assume that all switches participate as NMPs.

3) Communication Pattern: We evaluate all-to-all communication pattern; in the Fat Tree topology, we select two hosts at random and route the flow on a shortest path between them. In GEANT, we randomly choose two switches and route the flow between them.

4) Routing: We use shortest path routing and select one of the shortest paths at random for each flow when there are multiple possible shortest paths between its source and destination.

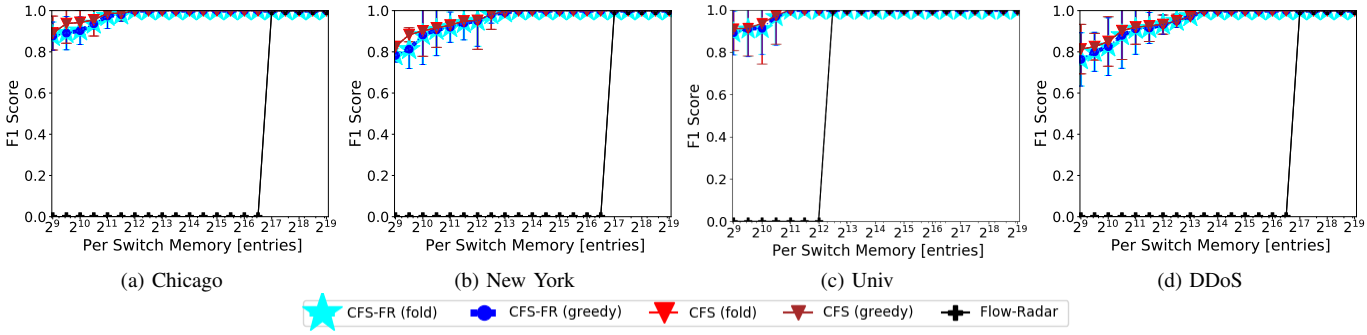
5) Datasets: We used the following datasets:

(a) The CAIDA Anonymized Internet Trace 2016 [1], from the Equinix-Chicago high-speed monitor, denoted Chicago.
(b) A data center trace [22], denoted by Univ.
(c) The CAIDA Anonymized Internet Trace 2018 from New York City, denoted by New York [2].



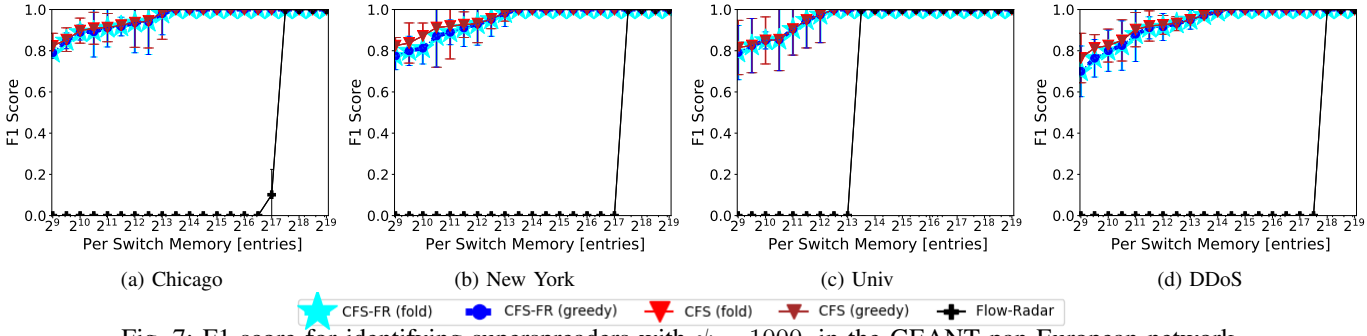
(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 5: Flow coverage when varying the per switch memory for the GEANT pan-european network.



(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 6: F1 score for identifying superspreaders with $\psi = 1000$, in the Fat Tree network topology (K=8).



(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 7: F1 score for identifying superspreaders with $\psi = 1000$, in the GEANT pan-European network.

(d) A Cyber attack trace from [3], denoted DDoS.

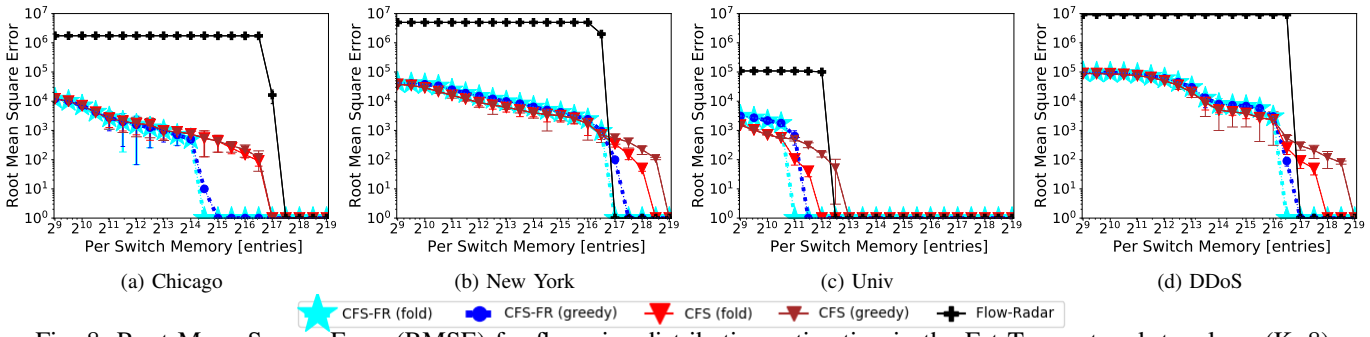
Each flow in the input packet trace is routed on a path which is selected as explained above.

B. Experimental Evaluation

1) Number of Flows and Flow Coverage Trade-off: Figure 3 shows results for synthetic traces where each flow has a single packet, and we vary the number of flows for a fixed number of entries per NMP. As illustrated,the attained flow coverage depends on the number of flows. Notice that Flow-Radar performs very poorly when there are too many flows whereas all our algorithms are near optimal for the entire range. Intuitively, Flow-Radar's network decode procedure fails when there are too many flows, while our cache based approach monitors as many flows as it can. Notice that CFS-FR in both fold and greedy approaches intersects with the optimal line and that CFS intersects with the optimal in the Fat-Tree topology, but is slightly sub-optimal in the GEANT topology, yet the fold approach is slightly better than the greedy approach.

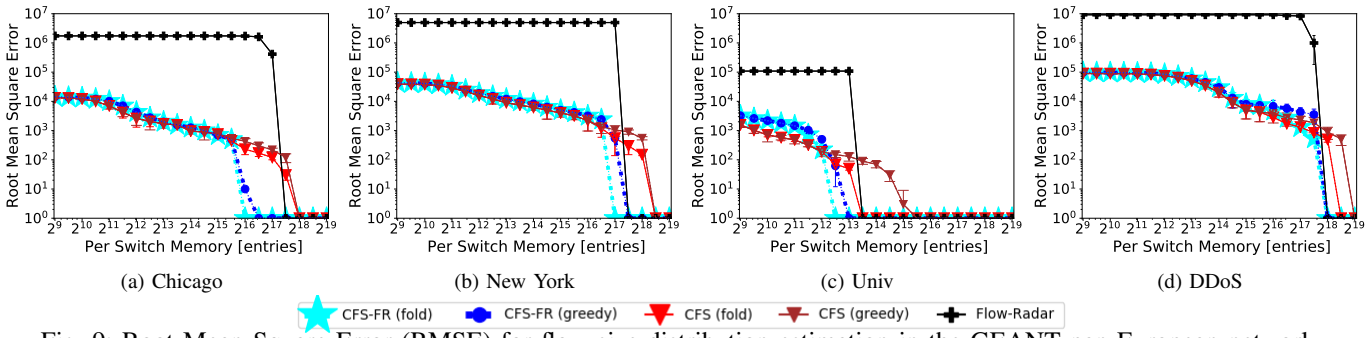
2) Per NMP Space and Flow Coverage Trade-off: Next, we vary the number of per-NMP flow entries, and study the attained flow coverage in real network traces. In the Fat Tree topology (Figure 4), notice that all our algorithms are near-optimal for the entire range. The CFS-FR approach is slightly worse than the CFS approach when there are too many entries, since the 10% memory allocated for Flow-Radar is useless for the range. Yet, it is slightly better when there is enough memory, and closely follows the optimal coverage all the way to full coverage. In contrast, the CFS method struggles to move from very high coverage (≈ 0.95) to full flow coverage. where the fold method, is slightly better than the greedy method. Flow-Radar has two phases, either it is useless when there is not enough space or it attains a full measurement when there is enough space. Also, note that DUS-HH only captures the heavy hitter flows which explains its low flow coverage.

Figure 5 shows the results for the GEANT pan-European network topology. As can be observed, the trends in this topology are very similar to those of the Fat-Tree topology despite the fundamental difference in network structure. However, observe that the number of required per-NMP flow entries also depends on the network topology (e.g., Flow-Radar requires slightly more space in GEANT than in Fat Tree).



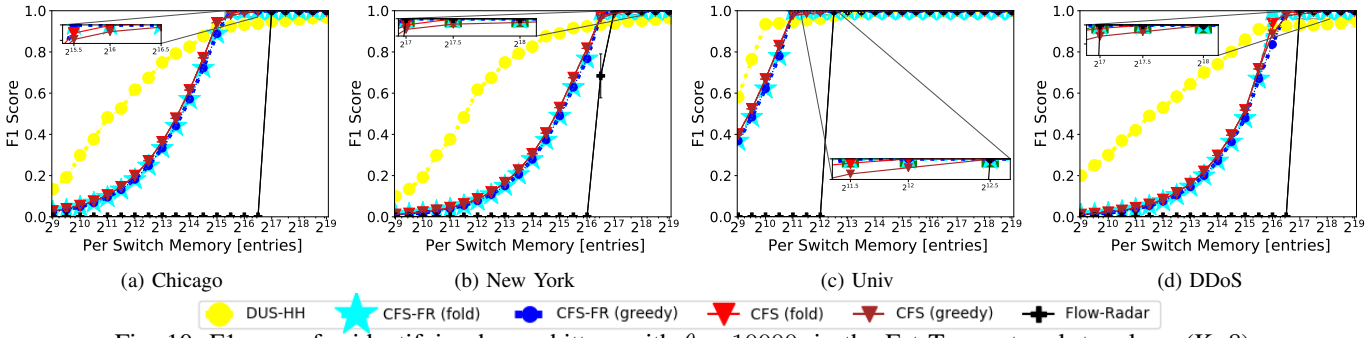
(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 8: Root Mean Square Error (RMSE) for flow size distribution estimation in the Fat Tree network topology (K=8).



(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 9: Root Mean Square Error (RMSE) for flow size distribution estimation in the GEANT pan-European network.



(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 10: F1 score for identifying heavy hitters with $\theta = 10000$, in the Fat Tree network topology (K=8).

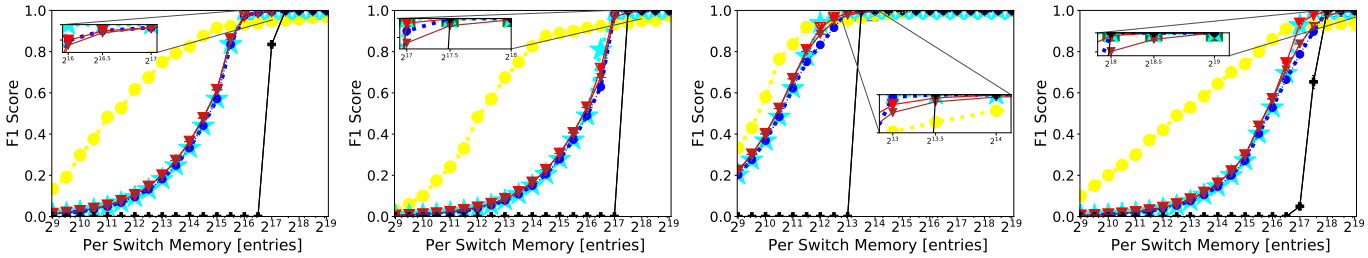
3) Superspreaders: Next, we evaluate the performance of algorithms when identifying superspreaders. Such measurements are often used by intrusion detection systems [56]. Figure 6 illustrates the results for the Fat-Tree topology, and Figure 7 for the GEANT network. Here all the variants of our algorithms achieve similar and considerably better performance than Flow-Radar. CFS approaches are slightly better than CFS-FR since Flow-Radar is inefficient for this task. Intuitively, we can identify the vast majority superspreaders even when do not monitor every single flow. Thus, our approaches are considerably more robust than Flow-Radar. Specifically, our algorithms reduce the needed space for the task by at least 50%, and up to 97%.

4) Flow Size Distribution: We now evaluate the root mean square error (RMSE) when estimating the flow size distribution on the Fat Tree (Figure 8), and on the GEANT network (Figure 9). First, observe that Flow-Radar is accurate when it has enough memory and is otherwise useless while CFS and CFS-FR are useful for the entire range. Notice that in some cases Flow-Radar outperforms CFS for some of the range. This is mainly because Flow-Radar can be 100% accurate when CFS has very high (but not perfect) coverage. However, CFS-FR is better than Flow-Radar for the entire range.

5) Heavy Hitters: We now focus on identifying the heavy hitter flows that transmit more than $\theta = 10000$ packets. Figure 10 shows results for the Fat Tree topology and Figure 11 shows results for the GEANT topology. First observe that when space is limited DUS-HH is better than all algorithms, as it efficiently monitor just the heavy hitter flows. When there is enough space, the flow based algorithms are better as they provide near exact measurement. Here, CFS and CFS FR out perform Flow-Radar for the entire range.

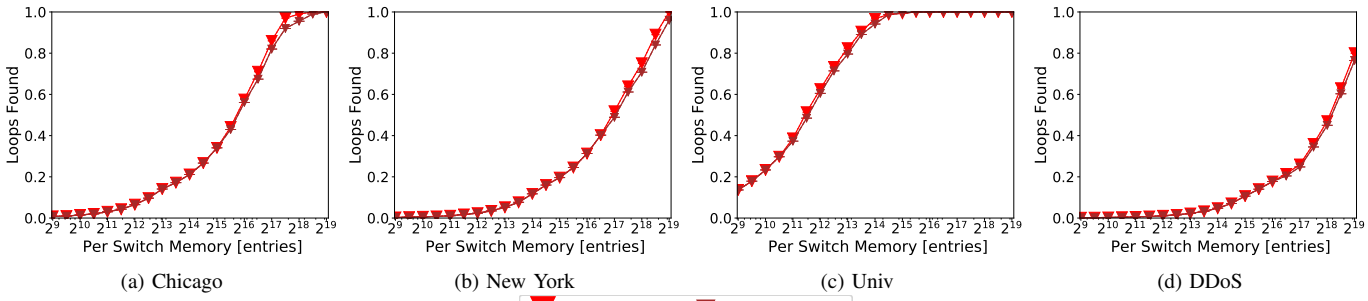
6) Loop detection: In the next experiment, we inject random routing loops between two of the NMPs on the flow's path. We evaluate how many of these potential loops are detected by CFS when varying the per-switch memory. One fundamental benefit of CFS compared to the related work is its ability to report routing loops in real time.

Figure 12 shows results for the Fat-Tree topology, while Figure 13 shows the results for the GEANT network. We detect more loops as we increase the amount of per-switch



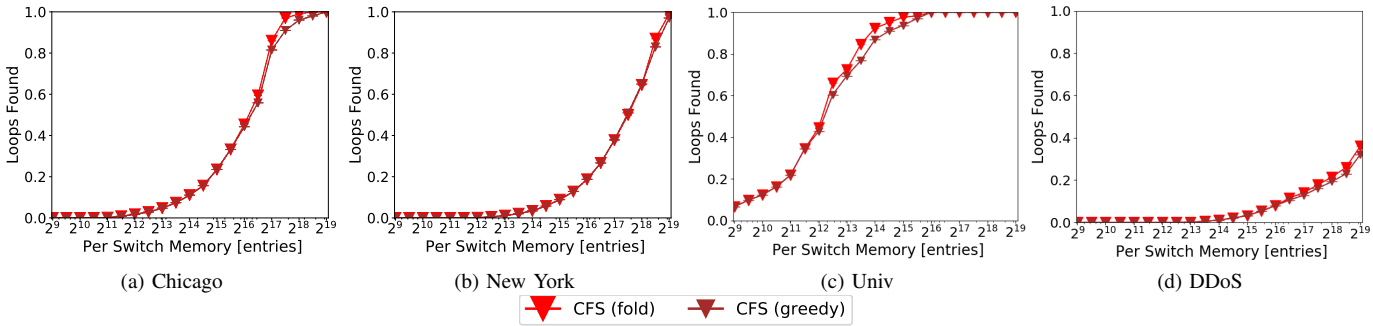
(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 11: F1 score for identifying heavy hitters with $\theta = 10000$, in the GEANT pan-European network.



(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 12: Portion of detectable loops in the Fat Tree network topology. (K=8).



(a) Chicago (b) New York (c) Univ (d) DDoS

Fig. 13: Portion of detectable loops in the GEANT pan-European network.

resources, and require more space for loop detection than for providing full flow coverage. We require more resources because not all the NMPs on the flow's path witness the loop.

V. Discussion

Our work shows that per-flow monitoring is an attractive measurement option for numerous network-wide measurement tasks including identifying the heavy hitters [9], hierarchical heavy hitters [16], estimating the flow size distribution [11], estimating the number of distinct flows [33], identifying superspreaders [56], and detecting routing loops. However, the previous flow monitoring algorithms either require prior knowledge about the underlying traffic [50] or require large amounts of resources (e.g., memory) to obtain full coverage.

Our work suggests cache-based algorithms where NMPs decide which flows to monitor without explicit coordination. Our mechanisms exploit the TTL field of the IP packet headers to break the symmetry and increase the total number of monitored flows. Specifically, we showed that our algorithms obtain near-optimal flow coverage for two network topologies and four real internet traces.

Next, we evaluated several network measurement tasks and showed that our algorithms indeed cash in on the better flow coverage. Specifically, we outperform Flow-Radar [42] for identifying superspreaders, estimating the flow size distribution, and for identifying the heavy hitters. When identifying superspreaders, we allow for a high F1 score (e.g., > 0.8) with less than 1% of the per NMP space required by Flow-Radar to match our accuracy. For detecting routing loops, CFS provides real-time detection, whereas Flow-Radar only detects routing loops at the end of the measurement.

We paid attention to the native primitives of modern network devices [6], and to the programming limitations of Protocol Independent Switch Architecture (PISA) devices. In the future, we seek to deploy CFS on such switches. Finally, we released the code we used [4] to benefit the community.

Acknowledgements: We thank the reviewers and our shepherd, Patrick P.C. Lee, for valuable feedback. This work was partially supported by the Zuckerman Institute and the Lynne and William Frankel Center for Computing Science, and the the Cyber Security Research Center at BGU.

REFERENCES

[1] The caida ucsd anonymized internet traces 2016 - january. 21st.

[2] The caida ucsd anonymized internet traces 2018 - january. 21st.

[3] Capture traces from mid-atlantic ccdc 2012. http://www.netresec.com/?page=MACCDC.

[4] CFS and CFS-FR implementation. https://github.com/Bilal-Tayh/CFS.

[5] *fat-tree topology*. https://en.wikipedia.org/wiki/Fattree.

[6] *Understanding mlx5 ethtool Counters*. https://community.mellanox.com/s/article/understanding-mlx5-ethtool-counters.

[7] *GEANT topology*, 29/03/2012. http://www.topology-zoo.org/files/Geant2012.gml.

[8] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. Detecting heavy flows in the SDN match and action model. *Computer Networks*, 2018.

[9] Daniel Anderson, Pryce Bevan, Kevin Lang, Edo Liberty, Lee Rhodes, and Justin Thaler. A high-performance algorithm for identifying frequent items in data streams. In *ACM IMC*, 2017.

[10] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM*, 2016.

[11] E. Assaf, R. B. Basat, G. Einziger, and R. Friedman. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *IEEE INFOCOM*, 2018.

[12] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *RANDOM*, 2002.

[13] Ran Ben Basat, Gil Einziger, Shir Landau Feibish, Jalil Moraney, and Danny Raz. Network-wide routing-oblivious heavy hitters. In *ACM ANCS*, 2018.

[14] Ran Ben Basat, Gil Einziger, and Roy Friedman. Give me some slack: Efficient network measurements. *Theoretical Computer Science*, 2019.

[15] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Optimal elephant flow detection. In *IEEE INFOCOM*. IEEE, 2017.

[16] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo Caggiani Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. *ACM SIGCOMM*, 2017.

[17] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo Caggiani Luizelli, and Erez Waisbard. Volumetric hierarchical heavy hitters. In *IEEE MASCOTS*, 2018.

[18] Ran Ben Basat, Gil Einziger, Junzhi Gong, Jalil Moraney, and Danny Raz. q-max: A unified scheme for improving network measurement throughput. In *ACM IMC*, 2019.

[19] Ran Ben Basat, Roy Friedman, and Rana Shahout. Stream frequency over interval queries. *Proc. VLDB Endow.*, 2018.

[20] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *IEEE INFOCOM*, 2017.

[21] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic In-Band Network Telemetry. In *ACM SIGCOMM*, 2020.

[22] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.

[23] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *ACM CoNEXT*, 2011.

[24] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[25] Christian Callegari, S Giordano, and Michele Pagano. An information-theoretic method for the detection of anomalies in network traffic. *Computers and Security*, 70, 07 2017.

[26] Min Chen, Shigang Chen, and Zhiping Cai. Counter tree: A scalable counter architecture for per-flow traffic measurement. *IEEE TON*, 2017.

[27] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. Catching the microburst culprits with snappy. In *SelfDN*, 2018.

[28] Graham Cormode. Continuous distributed monitoring: A short survey. In *AlMoDEP*, 2011.

[29] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 2008. Code: www.research.att.com/ marioh/frequent-items.html.

[30] Xenofontas Dimitropoulos, Paul Hurley, and Andreas Kind. Probabilistic lossy counting: An efficient algorithm for finding heavy hitters. *SIGCOMM CCR*, 2008.

[31] Gero Dittmann and Andreas Herkersdorf. Network processor load balancing for high-speed links. In *Proc. of the 2002 Int. Symp. on Performance Evaluation of Computer and Telecommunication Systems*.

[32] Nick G. Duffield, Carsten Lund, and Mikkel Thorup. Flow sampling under hard resource constraints. In *ACM SIGMETRICS*, 2004.

[33] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.

[34] Pedro Garcia-Teodoro, Jesús E. Díaz-Verdejo, Gabriel Maciá-Fernández, and E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Comp. and Security*, 2009.

[35] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven network telemetry. *ACM SIGCOMM*, 2018.

[36] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *SOSR*, 2018.

[37] Nicolas Hohn and Darryl Veitch. Inverting sampled traffic. In *ACM IMC*, 2003.

[38] Nan Hua, Bill Lin, Jun (Jim) Xu, and Haiquan (Chuck) Zhao. Brick: A novel exact active statistics counter architecture. In *ACM ANCS*, 2008.

[39] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, 2017.

[40] Jaeyeon Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE S&P*, 2004.

[41] Y. Li, H. Wu, T. Pan, H. Dai, J. Lu, and B. Liu. Case: Cache-assisted stretchable estimator for high speed per-flow measurement. In *IEEE INFOCOM*, 2016.

[42] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *USENIX NSDI*, 2016.

[43] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *ACM CoNEXT*, 2016.

[44] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, 2019.

[45] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[46] B. Mukherjee, L.T. Heberlein, and K.N. Levitt. Network intrusion detection. *Network, IEEE*, 1994.

[47] James B Orlin. Max flows in o (nm) time, or better. In *ACM STOC*, 2013.

[48] Ran Ben Basat, Xiaoqi Chen, Gil Einzinger, Ori Rottenstreich. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE ICNP*, 2018.

[49] Pegah Sattari. Revisiting IP Traceback as a Coupon Collector's Problem. In *PhD Dissertation*. University of California, Irvine, 2007.

[50] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. Csamp: A system for network-wide flow monitoring. In *USENIX NSDI*, 2008.

[51] Vyas Sekar, Michael K. Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *ACM IMC*, 2010.

[52] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR*, 2017.

[53] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *ACM SIGMOD*, 2020. Full version: https://arxiv.org/abs/2004.05076.

[54] P. Tune and D. Veitch. Sampling vs sketching: An information theoretic comparison. In *IEEE INFOCOM*, 2011.

[55] D. Veitch and P. Tune. Optimal skampling for the flow size distribution. *IEEE Transactions on Information Theory*, 61(6), June 2015.

[56] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *USENIX NSDI*, 2013.

[57] Haiquan Zhao, Ashwin Lall, Mitsunori Ogihara, and Jun Xu. Global iceberg detection over distributed data streams. In *IEEE ICDE*, 2010.

[58] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, 2015.