Memento: Making Sliding Windows Efficient for Heavy Hitters

Ran Ben Basat[1,2], Gil Einziger[3,4], Isaac Keslassy[2,5], Ariel Orda[2], Shay Vargaftik[2], Erez Waisbard[4]
[1] *Harvard University* [2] *Technion* [3] *Ben Gurion University* [4] *Nokia Bell Labs* [5] *VMware*

ABSTRACT

Cloud operators require real-time identification of Heavy Hitters (HH) and Hierarchical Heavy Hitters (HHH) for applications such as load balancing, traffic engineering, and attack mitigation. However, existing techniques are slow in detecting new heavy hitters.

In this paper, we make the case for identifying heavy hitters through *sliding windows*. Sliding windows are quicker and more accurate to detect new heavy hitters than current interval based methods, but to date had no practical algorithms. Accordingly, we introduce, design and analyze the *Memento* family of sliding window algorithms for the HH and HHH problems in the single-device and network-wide settings. Using extensive evaluations, we show that our single-device solutions attain similar accuracy and are by up to 273× faster than existing window-based techniques. Furthermore, we exemplify our network-wide HHH detection capabilities on a realistic testbed. To that end, we implemented Memento as an open-source extension to the popular HAProxy cloud load-balancer. In our evaluations, using an HTTP flood by 50 subnets, our network-wide approach detected the new subnets faster, and reduced the number of undetected flood requests by up to 37× compared to the alternatives.

1 INTRODUCTION

Cloud operators require fast and accurate single-device and network-wide detection of *Heavy Hitters (HH)* (most frequent flows) and of *Hierarchical Heavy Hitters (HHH)* (most frequent subnets) to attain real-time visibility of their traffic. These capabilities are essential building blocks in network functions such as load balancing [4, 23, 34, 39], traffic engineering [15, 17] and attack mitigation [33, 42, 45, 46, 48].

Quickly identifying changes in the HH and HHH is a key challenge [47] and can have a dramatic impact on performance. For example, faster detection of HH flows allows load-balancing and traffic engineering solutions to respond to traffic spikes swiftly. For attack mitigation systems, quicker and more accurate detection of HHH subnets means that less attack traffic reaches the victim. This is particularly important for combating *Distributed Denial of Service (DDoS)* attacks on cloud services, as they become a growing concern with the increasing number of connected devices (*i.e.,* Internet of things) [29, 35].

In this work, we show that *sliding windows* are faster than interval based measurements in detecting new (hierarchical) heavy hitters. Unfortunately, the idea of using a sliding window for HHH was previously dismissed, as the existing sliding-window algorithms were "markedly slower and less space-efficient in practice", to quote [40]. Intuitively, this is because the sampling methods used for accelerating interval methods do not naturally extend to sliding windows, even for the simpler HH problem. As a result, the merits of sliding windows have not been properly evaluated. Moreover, sliding windows do not provide network-wide measurement capabilities, as opposed to interval approaches [3, 26, 51, 55]. Consequently, most applications that use HH or HHH rely on interval based measurements [25, 42, 45, 48].

Contributions. Our goal is to make sliding windows practical for network applications. Accordingly, we focus on the fundamental HH and HHH problems in both single-device and network-wide measurement scenarios. We then introduce the *Memento* family of four algorithms—one for each problem (*i.e.,* HH and HHH) and each measurement scenario (*i.e.,* single-device and network-wide). These are rigorously analyzed and provide worst-case accuracy guarantees. Moreover, in the network-wide setting, we maximize the accuracy guarantee given a per-packet control bandwidth budget.

Using extensive evaluations on real packet traces, we show that the Memento algorithms achieve speedups of up to 14× in HH and up to 273× in HHH when compared to existing sliding-window solutions. We also show that they match the speed of the fastest interval-based algorithm [7]. Our algorithms detect emerging (hierarchical) heavy hitters consistently faster than interval-based approaches, and their accuracy is similar to that of slower sliding-window solutions.

Next, we implement a proof-of-concept network-wide HH and HHH measurement system. The controller uses our network-wide algorithms, and the measurement points are implemented on top of the popular HAProxy cloud load-balancer, which we extended with capabilities to rate-limit traffic from entire subnets. We evaluate the achievable accuracy given a per-packet bandwidth budget for reporting measurement data to the control. We introduce new communication methods and compare them with a traditional approach. We create an HTTP flood attack from 50 subnets and show that the detection time is near-optimal while using a bandwidth budget of 1 byte per packet. For the same budget, our methods exhibit a reduction of up to 37× in the number of undetected flood requests compared to the alternative. Finally, we open-source the Memento algorithms and the HAProxy cloud load-balancer extension [1].

2 BACKGROUND

Streaming algorithms [43] are designed to process a stream (sequence) of elements (in our case, packets) while analyzing the underlying data. The main challenge of these algorithms is the sheer volume of the data that they are required to process, motivating space-efficient solutions that process elements extremely fast.

One of the most studied streaming problems is that of identifying the *Heavy Hitters* (HH) (*i.e., elephant flows*) – the elements that frequently appear in the data. For instance, Space Saving (SS) [38] is a popular HH algorithm. SS utilizes a set of m counters, each associated with a key (flow identifier) and a value. Whenever a packet arrives, SS increments the value of its flow's counter if it exists. Otherwise, if there is a free counter, it is allocated to the flow with a value of 1. Finally, if no available counter exists, SS replaces the identifier of the flow with the smallest value with that of the current flow and increments its value. For example, assume that the minimal counter is associated with flow x and has a value of 4, while flow y does not have a counter. If a packet of flow y arrives, we will

reallocate x's counter to y and set its value to 5, leaving x without a counter. When queried for the value of flow z, SS returns the value of the counter associated with z, or the value of the minimal counter if there is no counter for z.

SS runs on *intervals*, *i.e.,* it estimates the flow sizes from the beginning of the measurement and is often reset to allow its data to be fresh [42]. Another way for analyzing only the recent data is to use a sliding window algorithm [22] in which an answer to a query only reflects the last W packets. WCSS [11] extends Space Saving to sliding windows, and achieves constant update and query time. Unfortunately, WCSS is too slow to keep up with line rates, and it serves as a baseline in this work. We expand on WCSS in the full version of the paper [9].

Hierarchical Heavy Hitters (HHH) are a generalization of the HH problem in which we identify frequent IP *networks*. That is, rather than looking for the large flows, we look for the networks whose aggregated volume exceeds a predetermined threshold. To that end, MST [41] proposed to utilize SS for tracking all networks. Specifically, it uses one SS instance for each network size and whenever a packet arrives, it computes all its prefixes (specifically H updates, as the size of the hierarchy) MST has two main drawbacks: first, it makes multiple SS updates, while even a single update may be too slow for keeping up with line rates; second, it solves the problem on intervals rather than on sliding windows. To alleviate the first problem, RHHH [7] proposes to draw a single random integer i uniformly between 1 and V (for a parameter $V \geq H$). If $i \leq H$ then RHHH makes a *single* SS update to the i'th prefix, and otherwise it ignores the packet. For example, for the above packet, $i = 7$ would be ignored while $i = 3$ would lead the algorithm to feed 181.0.0.0 into the relevant SS instance. While RHHH is fast enough to keep up with the line rate, its approach does not seem to extend to sliding windows easily, a gap we close in this paper.

3 WHY SLIDING WINDOWS?

We argue that once a new heavy hitter emerges, the sliding window method identifies it most quickly and accurately. Therefore, network applications that capitalize on sliding windows can potentially react faster to changes in the traffic. For simplicity, we consider accurate measurements, but the results are also valid for approximate measurements.

Window vs. interval. We start by comparing *sliding windows* to the *Interval* method that is commonly used in HHH-based DDoS mitigation systems [42, 45, 46, 48]. As depicted in Figure 1a, the Interval method relies on sequential interval measurements. Usually, the measurement data is available at the end of each measurement interval, wherein the *improved* Interval method it is accessible throughout each measurement period. There are two possible failure modes, namely: failing to detect a new heavy hitter (false negative), or falsely declaring a heavy hitter (false positive). Algorithms that follow the (improved) Interval method would either have false positives or false negatives. In contrast, sliding windows can avoid both errors. To show this, we start with the following definitions.

Definition 3.1 (Window Frequency). We denote by f_x^W the *window frequency* of flow x, *i.e.,* the *number* of packets transmitted by x over the last W packets.

Definition 3.2 (Normalized Window Frequency). We denote by $\frac{f_x^W}{W}$ the *normalized window frequency* of flow x, *i.e.,* the *fraction* of x's packets within the last W packets.

Next, window heavy hitters are flows whose normalized window frequency is larger than a user-defined threshold:

Definition 3.3 (Window Heavy Hitter). Flow x is a *window heavy hitter* if its normalized window frequency $\frac{f_x^W}{W}$ is larger than θ, where $\theta \in (0, 1)$ is a user-defined threshold.

Window optimality. The optimal detection point for new window heavy hitters is simply once their normalized window frequency is above a user-defined threshold. Reporting a flow earlier is *wrong* (false positive), and reporting it afterwards is *(too) late*. This means that sliding window measurements, *by definition*, have an optimal detection time.

Motivation. We motivate the definition for window heavy hitters with an experimental scenario where a new flow appears during the measurement and consumes, at a constant rate, a larger-than-the-threshold portion of the traffic after its initial appearance. We measure how long it takes for each measurement method to identify the new heavy hitter and evaluate the following measurement methods.

(i) *Interval*: The window frequency of each flow is estimated at the end of every measurement. This method represents limitations of sampling techniques (*e.g.,* [7, 24]) that require time to converge and thus cannot provide estimates during the measurement. (ii) *Improved interval*: Same as *interval*, but flow frequencies are estimated upon the arrival of each packet. This represents the best case scenario for the Interval method. (iii) *Window*: Sliding window, where frequencies are estimated upon packet arrivals.

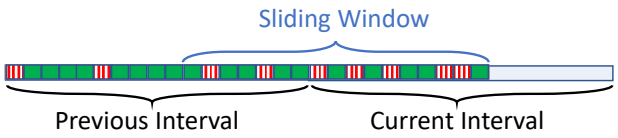
Figure 1b plots the detection time for each method as a function of the normalized frequency of the new heavy hitter. Intuitively, larger heavy hitters are detected faster, because less time passes before their normalized window frequency reaches the threshold. Indeed, the *sliding window* approach is always faster than the *Interval* and *improved Interval* methods. When the frequency is close to the detection threshold, we get up to 40% faster detection time compared to the Interval method. At the end of the tested range, sliding windows are still over 5% quicker. The Interval method is the slowest, as it estimates frequencies only at the end of the measurement. Thus, such a usage pattern is undesired for systems such as load balancing and attack mitigation.

4 SLIDING WINDOW ALGORITHMS

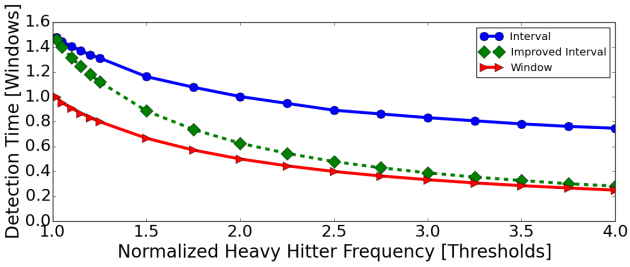
Our next step is to make sliding windows accessible to cloud operators. We do so by first introducing new single-device algorithms that are significantly faster than existing techniques, and then extend them to efficient network-wide algorithms that combine information from many measurement points to obtain a global perspective.

4.1 Heavy Hitters on Sliding Windows

Natural approach. Our goal is to produce faster sliding window algorithms. Intuitively, one can accelerate the performance of a heavy hitter algorithm by sub-sampling the packets. That is, we would like to sample packets with a probability of τ, use an HH algorithm with

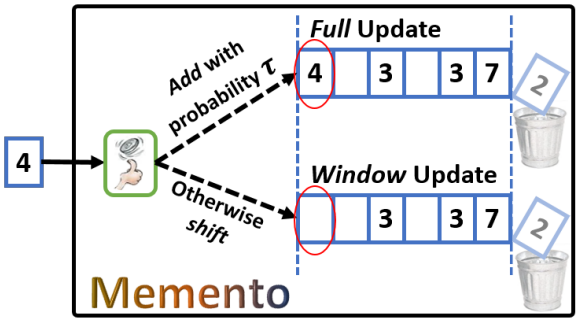


(a) An example of the periodic *interval* and *sliding window* methods. In this scenario, consider a threshold of nine packets. The solid-green flow is a window heavy-hitter as it has ten packets within the sliding window. However, the measurement interval method does not detect the green flow, as it only has five packets within the current interval (false negative). Intuitively, one can identify the green flow by lowering the threshold to 4 packets, but in that case, the striped red flow is detected as well (false positive).



(b) Effect of a new heavy hitter's frequency on its detection time. The x-axis is the ratio of the normalized heavy hitter's frequency and the user-defined threshold. The y-axis is the expected detection time in windows. For instance, when the frequency is twice the threshold, it takes a window algorithm half a window to detect the new heavy hitter whereas interval-based algorithms require between 0.6-1.0 windows.

Figure 1: Sliding windows compared to intervals.



(a) Our HH algorithm, Memento, utilizes two update methods: a slow Full update, and a faster Window update that only updates the sliding window. Speedup is achieved by performing Full updates for a small fraction of the packets. Here, the flow id 4 is inserted following the coin flip and 2 leaves the window (regardless of the coin flip).

(b) Our HHH algorithm, H-Memento, simply updates Memento with a single random prefix, achieving constant time complexity.

Figure 2: High-level overview of our algorithms.

a window size of $W \cdot \tau$ packets, and then multiply its estimations by a factor of τ^{-1}. Unfortunately, this does not yield the desired outcome as the number of samples from the window varies whereas sliding-window HH algorithms are designed for fixed-sized windows. Specifically, since the actual number of samples from the W sized window is distributed $\text{Bin}(W, \tau)$, this approach results in an additional error of $\pm\Theta\left(\sqrt{W(1-\tau)\cdot\tau^{-1}}\right)$ in the size of the reference window. Since we are interested in small values of τ to achieve speedup (see Section 6.3), this approach results in a considerable error.

Memento overview. Existing algorithms (*e.g.,* [10, 11, 32]) for identifying the heavy hitters over a sliding window are deterministic and thus have to process each and every packet fully. This processing, while running in asymptotic constant time, is computationally inefficient in practice due to multiple hash computations and memory accesses per packet. Our observation is that these updates actually have two logical parts. First, the algorithms handle the "forgetting" of old data that leaves the window as new data arrives, and second, recording the current packet's flow in the data structure. When analyzing where the algorithm spends most of its time, we observed that the first part is extremely lightweight compared with the resources used for recording the packet.

Accordingly, the key idea behind *Memento* is to decouple the computationally expensive operation of updating a packet (*Full update*) from the lightweight operation of *Window update*. Specifically, for each packet, Memento performs the Full update operation with probability τ; otherwise, it makes the quicker Window update.

Therefore, Memento alternates between the fast Window updates and the slower Full updates. Full updates include (1) forgetting outdated data *and* (2) adding a new item to the measurement data structure. On the other hand, Window updates only involve the former item, namely forgetting outdated data. That is, Memento maintains a W-sized window but most of the packets within that window are missing. Thus, it attains speedup but avoids the additional error that is caused by uniform samples. The concept is exemplified in Figure 2a.

Implementation. For simplicity, we built Memento on top of an existing sliding window HH algorithm. This makes it easier to implement, verify, and then compare with the current approaches. We picked WCSS as the underlying algorithm [11], but our approach is general and works on other window algorithms as well (*e.g.,* [10, 32]). Intuitively, when $\tau = 1$, Memento becomes identical to WCSS as it performs a full update for each packet. That is, the difference between Memento and WCSS is that for a $(1-\tau)$ fraction of the packets Memento makes only a window update and thereby saves computation resources. We provide more background on the algorithmic implementation of WCSS and Memento in the full version of the paper [9].

4.2 Extending to Hierarchical Heavy Hitters

Hierarchical heavy hitters monitor subnets and flow aggregates in addition to individual flows. We start by introducing existing approaches for HHH measurements on sliding windows.

Existing approaches. In MST [41], multiple HH instances are used to solve the HHH problem. This design trivially extends to sliding

windows by replacing the HH building blocks with window algorithms (e.g., WCSS [11]). This was proposed by [41] but dismissed as impractical. Replacing the underlying algorithms with Memento is slightly better as we can perform Window updates to most instances. Unfortunately, the update complexity remains $\Omega(H)$ which may still be too slow. In contrast, H-Memento achieves constant time updates, matching the complexity of interval algorithms [7]. Another natural approach comes from the RHHH [7] algorithm. RHHH shares the same data structure as MST but randomly updates at most a single HH instance which allows for constant time updates. Additionally, it makes small changes to the query procedure to compensate for the sampling error and guarantees that (with high probability) it will have no false negatives. This method does not work for sliding windows, as each HH instance is updated a varying number of times and monitors a possibly different window.

H-Memento's overview. In H-Memento we differ from the lattice structure of RHHH and MST. That is, we maintain a single large Memento instance and use it to monitor all the sampled prefixes. Therefore, we use just one sliding window to measure all subnets, which the underlying Memento does in constant time. This approach also has engineering benefits such as code reuse, simplicity, and maintainability. The update procedure of H-Memento is illustrated in Figure 2b. Next, we proceed with notations and definitions for the HHH problem, which we later use to detail H-Memento.

HHH notations and definitions. For brevity, Table 1 summarizes the notations used in this work. We consider IP prefixes (*e.g.,* 181.∗). A prefix without a wildcard (e.g., 181.7.20.6) is called *fully specified*. The notation $\mathcal{U}$ is the domain of the fully specified items. A prefix p_1 *generalizes* another prefix p_2 if p_1 is a prefix of p_2. For example, 181.7.20.∗ and 181.7.∗ generalize the (fully specified) 181.7.20.6. The *parent* of a prefix is the longest generalizing prefix, *e.g.,* 181.7.∗ is 181.7.20.∗'s parent. Definition 4.1 formalizes this concept.

Definition 4.1 (Generalization). Let p, q be prefixes. We say that p *generalizes* q and denote $p \preceq q$ if for each dimension i, $p_i = q_i$ or $p_i \preceq q_i$. We denote the set of fully specified items generalized by p using $H_p \triangleq \{e \in \mathcal{U} \mid e \preceq p\}$. Similarly, the set of every fully specified item that is generalized by a set of prefixes P is denoted by: $H_P \triangleq \cup_{p \in P} H_p$. Moreover, denote $p \prec q$ if $p \preceq q$ and $p \neq q$.

Definition 4.1 also deals with the more general multidimensional case. For example, we can consider tuples of the form (source IP, destination IP). In that case, fully specified "prefixes" are fully determined in both dimensions, *e.g.,* $(\langle 181.7.20.6\rangle, \langle 208.67.222.222\rangle)$. Also, observe that "prefixes" now have two parents, *e.g.,* $(\langle 181.7.20.*\rangle, \langle 208.67.222.222\rangle)$ and $(\langle 181.7.20.6\rangle, \langle 208.67.222.*\rangle)$ are both parents to $(\langle 181.7.20.6\rangle, \langle 208.67.222.222\rangle)$.

The *size of the hierarchy (H)* is the number of different prefixes that generalize a fully specified prefix. Next, we look at a set of prefixes P and denote $G(p|P)$ as the set of prefixes in P that are most closely generalized by the prefix p. That is, $G(p|P) \triangleq \{h : h \in P, h \prec p, \nexists h' \in P \; s.t. \; h \prec h' \prec p\}$.

For example, consider the prefix $p =< 142.14.* >$ and the set $P = \{< 142.14.13.* >, < 142.14.13.14 >\}$, then we have $G(p|P) = \{< 142.14.13.* >\}$. The window frequency of a prefix p is the total sum of packets within the window that are generalized by p, *i.e.,* $f_p^W \triangleq \sum_{e \in H_p} f_e^W$. Note that each packet is generalized by H different prefixes. This motivates us to look at the *conditioned* (residual) frequency that a prefix p adds to a set of already selected prefixes P. The conditioned frequency is defined as: $C_{p|P} \triangleq \sum_{e \in H_{P \cup \{p\}} \setminus H_P} f_e$.

Symbol	Meaning
$\mathbb{S}$	The packet stream.
N	Current number of packets (the stream length).
W	The window size.
H	Size of the hierarchy.
τ	Sampling probability.
V	Sampling ratio for HHH, $V \triangleq \frac{H}{\tau}$.
S_x^i	Variable for the i'th appearance of a prefix x.
S_x	Sampled prefixes with id x.
S	Sampled prefixes from all ids.
$\mathcal{U}$	Domain of fully specified items.
ϵ, ϵ_s, ϵ_a	Overall, sample, algorithm's error guarantee.
δ, δ_s, δ_a	Overall, sample, algorithm's confidence.
θ	Threshold parameter.
$C_{q\|P}$	Conditioned frequency of q with respect to P.
$G(q\|P)$	Subset of P with the closest prefixes to q.
f_q	Frequency of prefix q
$\widehat{f_q^+}, \widehat{f_q^-}$	Upper and lower bounds for f_q.
Z	inverse CDF of the normal distribution .
$\mathcal{B}$	per-packet control bandwidth budget.
$\mathbb{O}$	the minimal header size (in bytes),
E	bytes required to report a packet.
m	number of measurement points.
b	number of samples in each report.
$\mathfrak{E}_b$	overall error in network-wide settings.

Table 1: Summary of notations

We denote by X_p^W the number of times prefix p is sampled in the window, $\widehat{X_p^W}^+$ is an upper bound on X_p^W and $\widehat{X_p^W}^-$ is a lower bound. The notation $V \triangleq \frac{H}{\tau}$ stands for the sampling rate of each specific prefix. We define:

$\widehat{f_p^W} \triangleq \widehat{X_p^W} V$ – an estimator for p's frequency.

$\widehat{f_p^W}^+ \triangleq \widehat{X_p^W}^+ V$ – an upper bound for p's frequency.

$\widehat{f_p^W}^- \triangleq \widehat{X_p^W}^- V$ – a lower bound for p's frequency.

We now define the *depth* of a prefix (or a prefix tuple). Fully specified items are of depth 0, their parents are of depth 1 and more generally, the parent of an item with depth x is of depth $x + 1$. L denotes the maximal depth; observe that this may be lower than H (*e.g.,* in 2D byte-hierarchies $H = 25$ and $L = 9$). Hierarchical heavy hitters are calculated by iterating over all fully specified items (depth 0). If their frequency is larger than a threshold of θW, we add them to the set HHH_0. Then, we go over all the items with depth 1 and if their *conditioned* frequency, with regard to HHH_0, is above θW, we **add** them to the set. We name the resulting set HHH_1 and repeat the process L times, until the set HHH_L contains the

Algorithm 1 H-Memento (W, ε_a, τ)

Initialization: $Memento.init(H \cdot \epsilon_a^{-1}, W, \tau \cdot H)$
1: **function** UPDATE(x)
2: $Memento.update(RandomPrefix(x))$
3: **function** OUTPUT(θ)
4: $HHH = \phi$
5: **for** Level $\ell = 0$ up to L **do**
6: **for** each p in level ℓ **do** ▷ Only over prefixes with a counter.
7: $\widehat{C_{p|HHH}} = \widehat{f_p^W}^+ + calcPred(p, HHH)$
8: $\widehat{C_{p|HHH}} = \widehat{C_{p|HHH}} + 2Z_{1-\delta}\sqrt{VW}$ ▷ Compensate for sampling
9: **if** $\widehat{C_{p|HHH}} \geq \theta N$ **then** $HHH = HHH \cup \{p\}$
10: **return** HHH

Algorithm 2 calcPred for one dimension

1: **function** CALCPRED(prefix p, set P)
2: $R = 0$
3: **for** each $h \in G(p|P)$ **do** $R = R - \widehat{f_h^W}^-$
4: **return** R

Algorithm 3 calcPred for two dimensions

1: **function** CALCPRED(prefix p, set P)
2: $R = 0$
3: **for** each $h \in G(p|P)$ **do** $R = R - \widehat{f_h}^-$
4: **for** each pair $h, h' \in G(p|P)$ **do**
5: $q = glb(h, h')$
6: **if** $\nexists h_3 \neq h, h' \in G(p|P), q \preceq h_3$ **then** $R = R + \widehat{f_q}^+$
7: **return** R

(exact) hierarchical heavy hitters. Unfortunately, we need space that is linear in the stream size to calculate exact HHH (and even plain heavy hitters) [44]. Hence, as done by previous work [7, 19–21, 41], we solve approximate HHH.

A solution to the approximate HHH problem is a set of prefixes that satisfies the *Accuracy* and *Coverage* conditions (Definition 4.2). Here, Accuracy means that the estimated frequency of each prefix is within acceptable error bounds and Coverage means that the conditioned frequency of prefixes not included in the set is below the threshold. This does not mean that the conditioned frequency of prefixes that are included in the set is above the threshold. Thus, the set may contain a small number of subnets misidentified as HHH (false positives).

Definition 4.2 (Approximate HHHs). An algorithm $\mathbb{A}$ solves $(\delta, \epsilon, \theta)$ - APPROXIMATE WINDOW HIERARCHICAL HEAVY HITTERS if it returns a set of prefixes P that, for an arbitrary run of the algorithm, satisfies:

Accuracy: If $p \in P$ then $\Pr\left(\left|f_p^W - \widehat{f_p^W}\right| \leq \varepsilon W\right) \geq 1 - \delta$.

Coverage: If $q \notin P$ then $\Pr\left(C_{q|P} < \theta W\right) \geq 1 - \delta$.

H-Memento's full description. A pseudo-code for H-Memento is given in Algorithm 1. The output method performs the HHH set calculation as explained for exact HHH. The calculation yields an approximate result as we only have an approximation for the frequency of each prefix. Thus, we conservatively estimate conditioned frequencies.

For two dimensions, we use the inclusion-exclusion principle [16] (Definition 4.3) to avoid double counting.

Definition 4.3. Denote by $glb(h, h')$ the greatest lower bound of h and h'. $glb(h, h')$ is a unique common descendant of h and h' s.t. $\forall p : (q \preceq p) \wedge (p \preceq h) \wedge (p \preceq h') \Rightarrow p = q$. If h and h' have no common descendants, $glb(h, h') = 0$

A pseudo code for the update method is given in Algorithm 1, which is the same for one and two dimensions. The difference between these is encapsulated in the CALCPRED method which uses Algorithm 2 for one dimension and Algorithm 3 for two. In two dimensions, $C_{p|HHH}$ is first set in Line 7 of Algorithm 1. Then, we remove previously selected descendant heavy hitters (Line 3, Algorithm 3) and finally we add back the common descendants (Line 6, Algorithm 3). The sampling error is accounted for in Line 8. Intuitively, our analysis shows which τ values guarantee that H-Memento solves the approximate HHHs problem.

4.3 Network-Wide Measurements

As Figure 3 illustrates, we now discuss a centralized controller that receives data from multiple clients and forms a network-wide view of the traffic (*e.g.,* network-wide HH or HHH). Similarly to [3, 26] we assume that there are several measurement points and that each packet is measured once. Our design focus is on two critical aspects of this system: (1) a *communication method* between the clients and the controller that conveys the information in a timely and bandwidth-efficient manner, and (2) a fast *controller algorithm*.

Formal model. First, we define a sliding window in the network-wide model as the last W packets that were measured somewhere in the network. Intuitively, we want the controller to analyze the traffic of the most recent W packets *in the entire network*, as observed by the measurement points. For example, we may want to monitor the last million packets in the entire network.

(1) Communication method. We now suggest three methods to communicate with the controller. For each method, the frequency of messaging with the controller is according to the bandwidth budget ($\mathcal{B}$). That is, smaller reports can be sent more frequently but also deliver less information.

Aggregation. Existing HH algorithms are often *mergeable*, *i.e.,* the content of two HH instances can be efficiently merged [5]. We are unaware of previous work that targets HHH, but since MST [41] and RHHH [7] use HH algorithms as building blocks then they can be merged as well. This capability motivates the *Aggregation* communication method. In this method, each client periodically transmits all the entries of its HH algorithm to the controller. Given enough bandwidth, this method is intuitively the most communication-efficient, as all data is transmitted. However, as each message is large, we infrequently send messages to meet the bandwidth budget, which creates inaccuracies.

Sample. Most network devices are capable of transmitting uniform packet samples to the controller. Motivated by this capability, the Sample method samples packets with a fixed probability τ, and sends a report to the controller once per τ^{-1} packets. Thus on average, each message contains a single sample. This information is enveloped by the usual packet headers that are required to deliver the packet in the

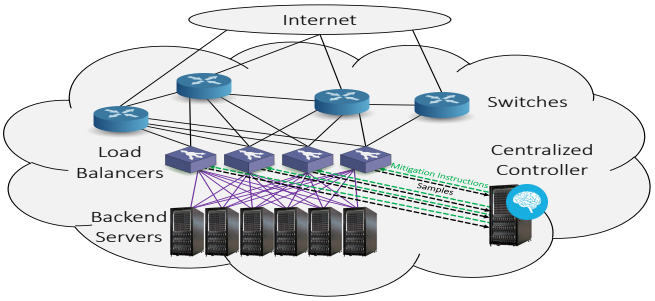


Figure 3: An overview of our system. The clients (load-balancers) perform the measurements and periodically send information to a centralized controller. The controller then runs a global sliding-window analysis. For example, in the case of an HTTP flood, it can mitigate the attack by instructing the clients which subnets to rate-limit or block.

network. We observe that this uses a significant portion of the bandwidth for the header fields of the transmitted packet. Yet, the Sample method is considerably easier to deploy than the Aggregate option, as the nodes only sample packets and do not run the measurement algorithms. The communication pattern is network-friendly as we get a stable flow of traffic from the clients to the controller.

Batch. The Batch approach is designed to utilize bandwidth more efficiently than the Sample. The idea is simple: instead of transmitting, on average, a single sample per message, we send on average b samples (*e.g.,* 100) per report. That is, we send a report once per $\tau^{-1}b$ packets, containing all the sampled packets within this period. This pattern utilizes the bandwidth more efficiently as the payload ratio of the message is considerably higher. However, it also creates delays in reporting new information to the controller. Our analysis is used to find the optimal batch size b and minimize the total error.

(2) Controller algorithm. The controller maintains an instance of Memento or H-Memento where we term the respective algorithms D-Memento and D-H-Memento. The controller behaves slightly differently in each option.

Aggregation. Aggregation is used in this study only as a baseline. Thus, instead of implementing a specific algorithm, we simulate an *idealized* aggregation technique with an unlimited space at the controller and no accuracy losses upon merging. As we later show, the Sample and Batch approaches outperform this Aggregation method; thus, we conclusively demonstrate that they are superior to *any* aggregation technique.

Sample and Batch. In the Sample and Batch schemes, the controller maintains a Memento or H-Memento instance. When receiving a report, it first performs a Full update for each sampled packet and then makes Window updates for the un-sampled ones. In total, the Sample performs τ^{-1} updates and the Batch performs $\tau^{-1}b$ updates.

5 ANALYSIS

This section is divided into two parts; first, Section 5.1 analyzes our single-device Memento and H-Memento algorithms and shows accuracy guarantees. Due to space limits, some proofs are omitted from this version and are available in the full version of this paper [9]. Next, Section 5.2 analyzes our network-wide D-Memento and D-H-Memento algorithms, and explains how to find the optimal batch size (in terms of guaranteed error) given a certain (per-packet) bandwidth budget.

5.1 Memento and H-Memento Analysis

This section surveys the main theoretical results for Memento and for H-Memento. These assure correctness as long as the sampling probability is large enough.

Formal model. Our traffic is modeled as a stream $\mathbb{S}$. It is initially empty, and a packet is added at each step. A sliding-window algorithm considers only the last W packets, denoted as $\mathbb{S}^W$. The notation f_e^W denotes the frequency of flow e in $\mathbb{S}^W$. Given e, a heavy hitters algorithm provides an estimator $\widehat{f_e^W}$ for f_e^W. We formalize the problem as follows:

Definition 5.1. An algorithm solves (ϵ, δ) - WINDOW FREQUENCY ESTIMATION if given a query for a flow (x), it provides $\widehat{f_x^W}$ such that $\Pr\left[\left|f_x^W - \widehat{f_x^W}\right| \leq \varepsilon W\right] \geq 1 - \delta$.

Memento. Theorem 5.2 is the main theoretical result for Memento. It states that Memento solves the (ϵ, δ) - WINDOW FREQUENCY ESTIMATION problem for $\varepsilon = \varepsilon_a + \varepsilon_s$ whenever it is allocated $O(1/\epsilon_a)$ counters and has a sampling probability that satisfies $\tau \geq Z_{1-\frac{\delta}{4}} W^{-1} \epsilon_s^{-2}$, where Z is the inverse of the cumulative density function of the normal distribution with mean 0 and standard deviation of 1. Note that $Z_{1-\frac{\delta}{4}}$ satisfies $Z < 4$ for any $\delta > 10^{-6}$. In other words, the theorem emphasizes the trade off between the amount of space allocated and the sampling rate, for achieving a target error bound ϵ. Specifically, if the algorithm has many counters (*i.e.,* have a low ϵ_a), then we can afford a higher ϵ_s (*i.e.,* the sampling rate can be low). The proof appears in the full version of this paper [9].

THEOREM 5.2. *Memento solves* (ϵ, δ) - WINDOWED FREQUENCY ESTIMATION *for* $\varepsilon = \varepsilon_a + \varepsilon_s$ *and* $\tau \geq Z_{1-\frac{\delta}{4}} W^{-1} \epsilon_s^{-2}$.

H-Memento. Theorem 5.3 is our main result for H-Memento. It says that H-Memento is correct for any $\tau > Z_{1-\frac{\delta}{2}} H W^{-1} \varepsilon_s{}^{-2}$, where H is the size of the hierarchic domain ($H = 5$ for source hierarchies and $H = 25$ for (source,destination) hierarchies). The proof is given in the full version of this paper [9].

THEOREM 5.3. *H-Memento solves* $(\delta, \epsilon, \theta)$ - APPROXIMATE WINDOWED HIERARCHICAL HEAVY HITTERS *for* $\tau \geq Z_{1-\frac{\delta}{2}} H W^{-1} \varepsilon_s{}^{-2}$.

5.2 D-Memento and D-H-Memento Analysis

We now provide analysis for our network-wide D-Memento and D-H-Memento algorithms. Intuitively, the error in these algorithms comes from two origins. First, an error due to *sampling*, which is quantified by Theorems 5.2 and 5.3. However, there is an additional error that is caused by the *delay in transmission*, as the measurement points only send the sampled packets once in every $b\tau^{-1}$ packets. If a measurement point has a low traffic rate, it may take long time for it to see $b\tau^{-1}$ packets; in this case, all of its samples may be obsolete and may not belong in the most recent window. Therefore, our first step is to reason about the accuracy impact of the Batch and Sample methods.

Notations and definitions. We denote the *bandwidth budget* as $\mathcal{B}$ bytes/packet. That is, $\mathcal{B}$ determines how much traffic is used for communicating between the measurement points and the controller. This communication is done using standard packets, which have header field overheads. We denote by $\mathbb{O}$ the minimal header size (in bytes) of the chosen transmission protocol (*e.g.,* 64 bytes for TCP). Next, reporting a sampled packet requires E bytes (*e.g.,* 4 bytes for srcip, or 8 bytes for (srcip,dstip) pair). We also denote by m the total number of measurement points.

Model. Intuitively, we can choose two (dependent) parameters: the sampling rate, τ, and the batch size b. That is, each measurement point samples with probability τ until it gathers b packets. At this point, it assembles an $(\mathbb{O}+Eb)$-sized packet that encodes the sampled packet and sends it to the controller. As the expected number of packets required to gather a b sized batch is $b\tau^{-1}$, the bandwidth constraint can be written as $(\mathbb{O}+Eb)/(b\tau^{-1}) \leq \mathcal{B}$. Specifically, this allows to express the maximum allowed sampling probability as $\tau = \mathcal{B}b/(\mathbb{O}+Eb)$ since sampling at a lower rate would not utilize the entire bandwidth and would result in sub-optimal accuracy.

Accuracy of the Batch and Sample methods. We can now quantify the error of the Batch and Sample methods. Intuitively, we have to factor the delays in communication (as we only report per a fixed number of packets to stay within the bandwidth budget). For example, if there are two measurement points in which one processes a million requests per second while the other only a thousand, the batches of the second point would include many obsolete packets that are not within the current window. However, recall that these reports only reflect $b\tau^{-1}$ packets at each of the m points. Therefore, we conclude that:

THEOREM 5.4. *The error created by the delayed reporting in the Batch method is bounded by* $mb\tau^{-1}$.

Next, Theorems 5.2 and 5.3 enable us to bound the sampling error as a function of τ, while Theorem 5.4 bounds the delayed reporting error. The following theorem applies for D-Memento (using $H = 1$) and for D-H-Memento (using the appropriate H value). As the round trip time inside the data center is small compared to window sizes that are of interest, the error caused by the delay of packet transmissions is negligible, and thus we do not factor it here. Theorem 5.5 quantifies the overall error in the Batch method; the error of the Sample method is derived when setting $b = 1$.

THEOREM 5.5. *Given overhead* $\mathbb{O}$*, batch size b, bandwidth budget* $\mathcal{B}$*, sample payload size S, window size W and confidence* δ_s*, the overall error* $\mathfrak{E}_b$ *(in packets) is at most:*

$$\mathfrak{E}_b = m(\mathbb{O}+Eb)/\mathcal{B} + \sqrt{HWZ_{1-\frac{\delta_s}{2}}(\mathbb{O}+Eb)/(\mathcal{B}b)}.$$

PROOF. According to Theorem 5.3, we have that $\epsilon_s = \sqrt{HW^{-1}Z_{1-\frac{\delta_s}{2}}\tau^{-1}}$. This means that our overall error is bounded by:

$$\begin{aligned}\mathfrak{E}_b &= bm\tau^{-1} + W\epsilon_s = bm\tau^{-1} + \sqrt{HWZ_{1-\frac{\delta_s}{2}}\tau^{-1}} \\ &= m(\mathbb{O}+Eb)/\mathcal{B} + \sqrt{HWZ_{1-\frac{\delta_s}{2}}(\mathbb{O}+Eb)/(\mathcal{B}b)}. \quad \square\end{aligned}$$

Formally, we showed a bound of $\mathfrak{E}_b$, for each choice of b. The guarantees for the Sample method are given by fixing $b = 1$. The next step is to use Theorem 5.5 to calculate the optimal batch size b given a bandwidth budget $\mathcal{B}$. Thus, we get the best achievable accuracy for the Batch method within the bandwidth limitation. We have:

$$\frac{\partial \mathfrak{E}_b}{\partial b} = mE/\mathcal{B} + \frac{HWZ_{1-\frac{\delta_s}{2}}(E/B - \mathbb{O}/(\mathcal{B}b^2))}{2\sqrt{(\mathbb{O}+Eb)/(\mathcal{B}b)}}.$$

We then compare this expression to zero to compute the optimal batch size b. This is easily done with numerical methods.

For example, for a TCP connection ($\mathbb{O} = 64$); ten measurement points ($k = 10$); source IP hierarchy ($E = 4, H = 5$); error probability of $\delta = 0.01\%$; a window of size $W = 10^6$; and a bandwidth quota of $\mathcal{B} = 1$ byte per packet, the optimal batch size is $b = 44$. The resulting (overall) error guarantee is 13K packets (i.e., an error of 1.3%). Increasing the bandwidth budget to $\mathcal{B} = 5$ bytes decreases the absolute error to 5.3K packets (0.53%) while increasing the optimal batch size to $b = 68$. When increasing the window size (W), the *absolute* error increases by an $O\left(\sqrt{W}\right)$ factor and the error (as a fraction of W) decreases. For example, increasing the window size to 10^7 increases the optimal batch size to $b = 109$, while reducing the error to 0.15%. Alternatively, 2D source/destination hierarchies (increasing H from 5 to 25) result in a slightly larger error and a higher optimal batch size.

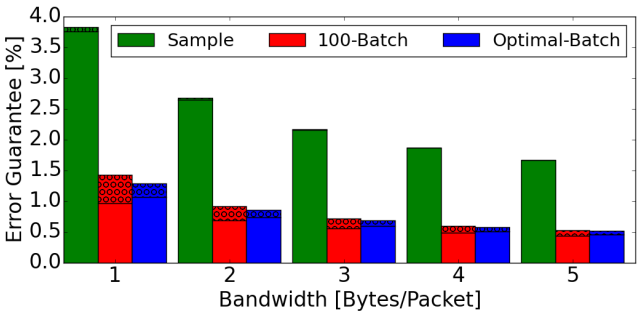


Figure 4: Comparing the accuracy *guarantees* of varying synchronization techniques. The parts hatched with circles quantify the bound on the error that is caused by the delayed synchronization.

Figure 4 illustrates the accuracy guarantee provided by each method. We compare three synchronization variants – Sample, Batch with $b = 100$, and Batch with an optimal b (varies with $\mathcal{B}$), as explained above. As depicted, Sample has the smallest delay error and yet provides the worst guarantees as it conveys little information within the bandwidth budget. The 100-Batch method has lower a sampling error (as its sampling rate is higher), but its reporting delay makes the overall error larger. For larger values of $\mathcal{B}$, the optimal batch size grows closer to 100 and the accuracy gap narrows.

6 EVALUATION

Server. Our evaluation was performed on a Dell 730 server running Ubuntu 16.04.01 release. The server has 128GB of RAM and an Intel Xeon CPU E5-2667 v4@ 3.20GHz.

Traces. We use real packet traces collected from an edge router *(Edge)* [2], a datacenter *(Datacenter)* [14], and a CAIDA backbone link *(Backbone)* [28].

Algorithms and implementation. For the HH problem, we compare Memento and WCSS [11]. For WCSS we use our Memento implementation without sampling ($\tau = 1$). For the HHH problem, we compare H-Memento to MST [41] and RHHH [7] (interval algorithms). We use the code released by the original authors of these algorithms. We also form the *Baseline* sliding window algorithm by replacing the underlying algorithm in MST [41] with WCSS. Specifically, MST proposed to use Lee and Ting's algorithm [36] as WCSS was not known at the time. By replacing the algorithm with the WCSS, a state of the art window algorithm, we compare with the best variant known today.

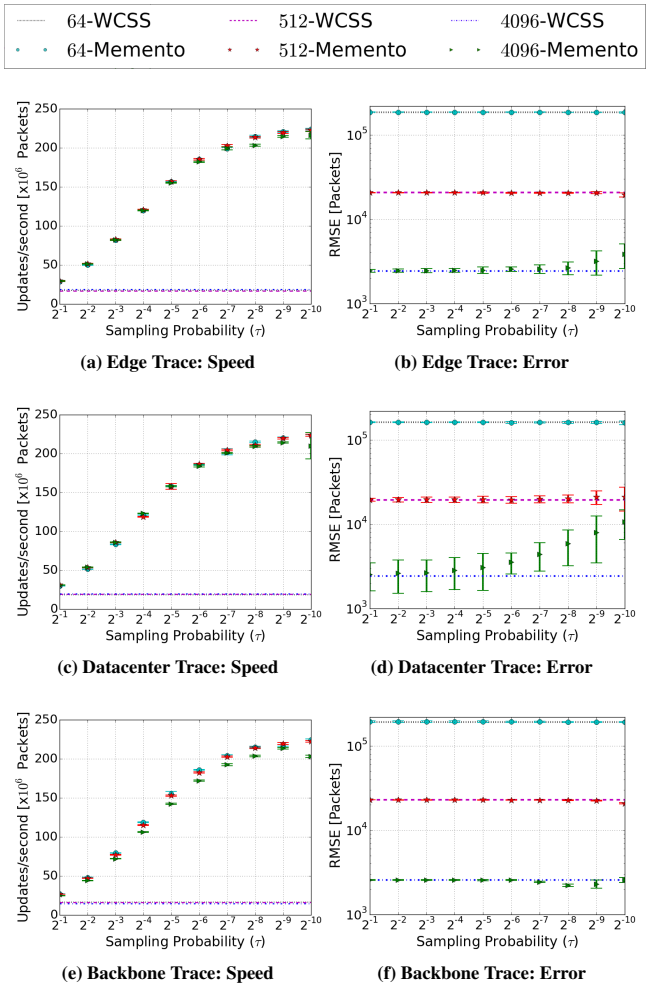
Yardsticks. We consider source IP hierarchies in byte granularity ($H = 5$) and two-dimensional source/destination hierarchies ($H = 25$). Such hierarchies are also used in [7, 21, 41]. We run each data point 5 times and use two-sided Student's t-tests to determine the 95% confidence intervals.

We evaluate the empirical error in the *On Arrival model* [11, 13], where for each packet we estimate its flow (denoted s_t) size. We then calculate the *Root Mean Square Error (RMSE), i.e., RMSE(Alg)* $\triangleq \sqrt{\frac{1}{|N|}\sum_{t=1}^{N}(\widehat{f_{s_t}} - f_{s_t})^2}$.

6.1 Heavy Hitters Evaluation

We evaluate the effect of the sampling probability τ on the operation speed and empirical accuracy of Memento, and use the speed and accuracy of WCSS as a reference point for this evaluation. The notation X-WCSS stands for WCSS that is allocated X counters (for X$\in \{64, 512, 4096\}$). Similarly, the notation X-Memento is for Memento with X counters. The window size is set to $W \triangleq 5$ million packets and the interval length is set to $N \triangleq 16$ million packets.

As depicted in Figure 5, the update speed is determined by the sampling probability and is almost indifferent to changes in the number of counters. Memento achieves a speedup of up to 14× compared to WCSS. As expected, allocating more counters also improves the accuracy. It is also evident that the error of Memento is almost identical to that of WCSS, which indicates that it works well for the range. The smallest evaluated τ, namely, $\tau = 2^{-10}$, already exhibits slight accuracy degradation, which shows the limit of our approach. It appears that a larger number of counters, or heavy tailed workloads (such as the Backbone trace), allow for even smaller sampling probabilities without a impact to the attained accuracy.



(a) Edge Trace: Speed (b) Edge Trace: Error

(c) Datacenter Trace: Speed (d) Datacenter Trace: Error

(e) Backbone Trace: Speed (f) Backbone Trace: Error

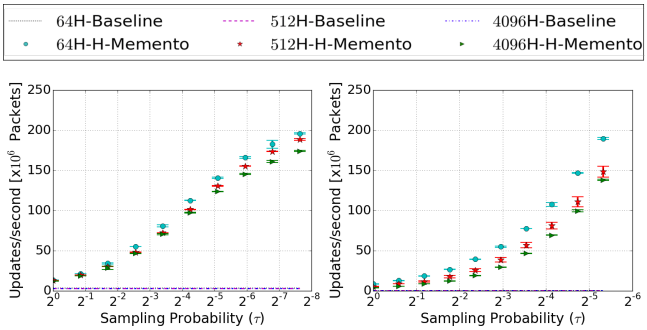
Figure 5: Effect of the sampling probability τ on the speed and accuracy for varying number of counters given three different traces and a window size $W = 5M$. *Memento* is considerably faster than WCSS, but the accuracy of both algorithms is almost the same despite *Memento's* use of sampling, except when the sampling rate is low and the number of counters is high. Even then, this is mainly evident in the skewed Datacenter trace.

6.2 Hierarchical Heavy Hitters Evaluation

H-Memento vs. existing window algorithm. Next, we evaluate H-Memento and compare it to the Baseline algorithm. We consider two common types of hierarchies, namely a one-dimensional source hierarchy ($H = 5$) and two-dimensional source/destination hierarchies ($H = 25$). Note that H-Memento performs updates in constant time while the Baseline does it in $O(H)$. Following the insights of Figure 5, we evaluate H-Memento with a sampling rate τ such that $\tau \geq H \cdot 2^{-10}$, so that each of the H prefixes is sampled with a probability of at least 2^{-10}. That is, we do not allow sampling probabilities of $\tau < H \cdot 2^{-10}$ to get an effective sampling rate of at least 2^{-10}, which is the range in which Memento is accurate.

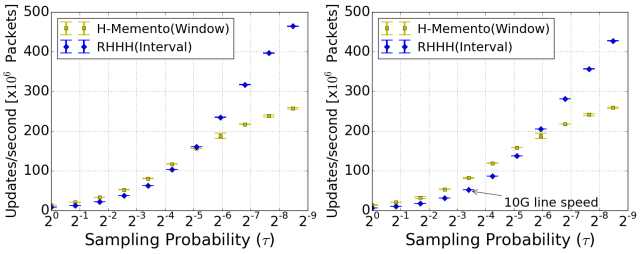
We evaluate three configurations for each algorithm, with a varying number of counters. The notation 64H denotes the use of $64 \cdot 5 = 320$ counters when $H = 5$, and 1600 counters when $H = 25$. The notations 512H and 4096H follow the same rule. In the Baseline algorithm, the counters are utilized in H equally-sized WCSS instances, while H-Memento has a single Memento instance with that many counters.

Figure 6 shows the evaluation results. We can see how τ is the dominating performance parameter. H-Memento achieves up to a 52× speedup in source hierarchies and a 273× speedup in



(a) Backbone trace - 1 D (H=5) (b) Backbone trace - 2 D (H=25)

Figure 6: Effect of the sampling probability on the speed of *H-Memento*, compared to the Baseline algorithm in the Backbone trace. Note that H-Memento achieves a speedup of up to 53× in 1D and up to 273× in 2D. Results for the Edge and Datacenter traces are similar.



(a) One dimension (H=5) (b) Two dimensions (H=25)

Figure 7: Speed comparison between RHHH (interval algorithm) and H-Memento (window algorithm) on the Backbone dataset. The annotated point shows the throughput of the ($\tau = 1/10$)-RHHH algorithm that is reported to meet the 10G line speed using a single core [7]. That is, H-Memento is slightly faster than RHHH in the parameter range of 10G lines.

source/destination hierarchies. This difference is explained by the fact that the Baseline algorithm makes H expensive Full updates for each packet, while H-Memento usually performs a single Window update.

H-Memento vs. interval algorithm. Next, we compare the throughput of H-Memento to the previously suggested RHHH [7]. H-Memento and RHHH are similar in their use of samples to increase performance. Moreover, RHHH is the fastest known interval algorithm for the HHH problem. Our results, presented in Figure 7, show that *H-Memento* is faster than RHHH for small sampling ratios. The reason lies in the implementation of the sampling. Namely, in RHHH, sampling is implemented as a geometric random variable, which is inefficient for small sampling probabilities, whereas in *H-Memento*, it is performed using a random number table. Still, as the sampling probability gets lower, the geometric calculation becomes more efficient, and eventually, RHHH is faster than *H-Memento*. This is because H-Memento performs a Window update for most packets, while RHHH only decrements a counter.

Looking at both performance figures independently, we conclude that *H-Memento* achieves very high performance and is likely to incur little overheads in a virtual switch implementation in a similar manner to RHHH.

6.3 Network-Wide Evaluation

This section describes our proof-of-concept system. We incorporated H-Memento into HAProxy which provides the capability to monitor traffic from subnets, an ability which we used to implement rate limiting for subnets. Our controller periodically receives information from (in the Batch, Sample or Aggregate method) the load-balancers and uses this to perform the HHH measurement (with the D-H-Memento algorithm). Then, the HHH output can be used as a simple threshold-based attack mitigation application where a subnet is rate-limited if its window frequency is above the threshold.

HAProxy. We implemented and integrated our algorithms into the open-source HAProxy load-balancer (version 1.8.1). Specifically, we leveraged and extended HAProxy's Access Control List (ACL) capabilities, to allow the updates of our algorithms with new arriving data as well as to perform mitigation (*i.e.,* Deny or Tarpit) when an attacker is identified.

Traffic generation. Our goal is to obtain realistic measurements involving multiple simultaneous stateful connections such as HTTP GET and POST requests from multiple clients towards the load-balancers. To that end, we developed a tool that enables a single commodity desktop to maintain and initiate stateful HTTP GET and POST requests sourcing from multiple IP addresses. Our solution requires the cooperation of both ends (*i.e.,* the traffic generators and the load-balancer servers) for an arbitrarily large IP pool.

It is based on the *NFQUEUE* and *libnetfilter*-queue Linux targets that enable the delegation of the decision on packets to a userspace software. As reported by the Apache *ab load* testing tool, using a single commodity computer, we can initiate and maintain up to 30,000 stateful HTTP requests per second from different IPs without using the HTTP keep-alive feature. We are only limited by the pace at which the Linux kernel opens and closes sockets (*e.g.,* TCP timeout).

Controller. We implemented in C a test controller that communicates with the load-balancers via sockets. It holds a local HHH algorithm implementation and exchanges information with the load-balancers (*e.g.,* receives aggregations, samples, or batches). The controller then generates a global and coherent window view of the ingress traffic.

Testbed. We built a testbed involving three physical servers. The first is used for traffic generation towards the load-balancers. Specifically, we used several apache ab instances augmented with our tool to generate realistic stateful traffic from multiple IP addresses with delay and racing among different clients. The second station holds ten autonomous instances (*i.e.,* separate processes) of HAProxy load-balancers listening on different ports for incoming requests. Finally, at the third station, we used docker to deploy Apache server instances listening on different sockets.

6.3.1 H-Memento's Accuracy. In this experiment, we evaluate MST (denoted as *Interval*), the Baseline algorithm and H-Memento with a single load-balancer client. Our goal is to monitor

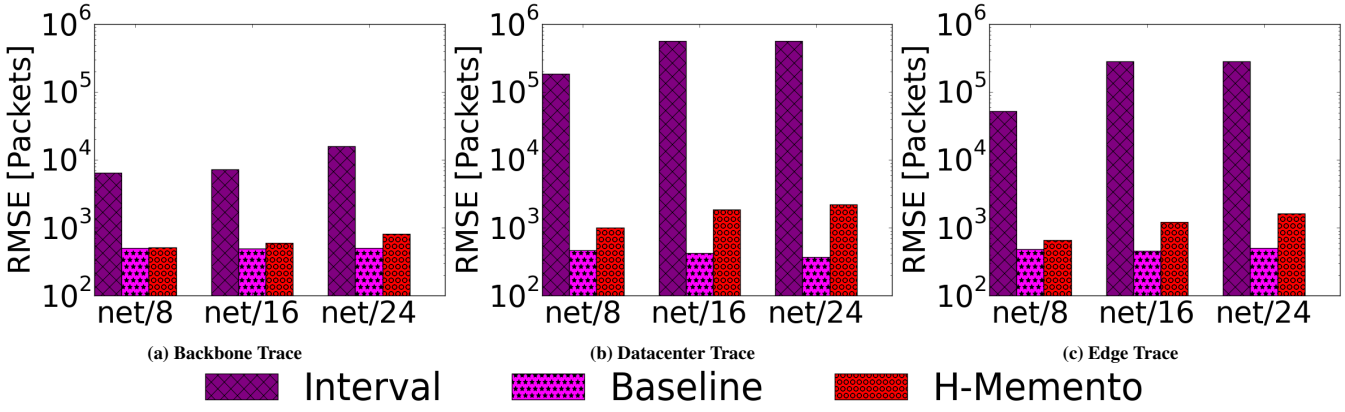


Figure 8: Comparing the error of H-Memento.

the last 1,000,000 HTTP requests that have entered the load-balancer. The Baseline algorithm and H-Memento are set at $\epsilon_a = 0.1\%$ and a window size of 1,000,000 requests. The MST Interval instance is using a measurement period of 1,000,000 requests and is configured with $\epsilon_a = 0.025\%$, resulting in comparable memory usage. For each new incoming HTTP request, each algorithm estimates the frequency of each of its IP prefixes.

The results are depicted in Figure 8. In all the traces, the Interval approach is the least accurate, while as expected, H-Memento is slightly less accurate than the Baseline algorithm due to its use of sampling. These conclusions hold for every prefix length and testbed workload.

6.3.2 Accuracy and Traffic Budget. In this experiment, we generate traffic towards ten load-balancers communicating with a centralized controller that maintains a global window view of the last 1,000,000 requests that entered the system. We evaluate the three different transmission methods (Aggregation, Sample, and Batch) with the same 1-byte per packet control traffic budget.

Results. Figure 9 depicts the results. As indicated, the best accuracy is achieved by the Batch approach, while Sample significantly outperforms Aggregation. Intuitively, the Aggregation method sends the largest messages, each of which contains the full information known to the measurement point. Its drawback is a long delay between controller updates. The Sample method has a smaller delay but utilizes the bandwidth inefficiently due to the packet header overheads. Finally, Batch has a slightly higher delay but delivers more data within the bandwidth budget, which improves the controller's accuracy.

6.4 HTTP Flood Evaluation

We now evaluate our detection system in an HTTP flood. Our deployment consists of ten HAProxy load-balancers that serve as the entry point and direct requests to Apache servers. The HAProxy load-balancers also report to the centralized controller that discovers subnets that exceed the user-defined threshold. The bandwidth budget is set to 1-byte per packet and the window size is $W = 1$ million packets.

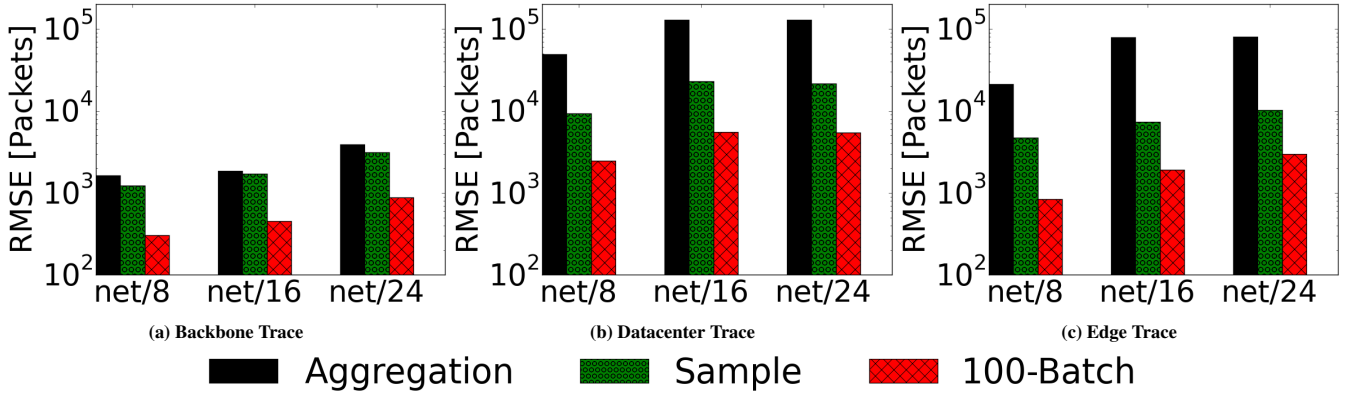


Figure 9: Network-wide evaluation. Accuracy attained by D-H-Memento with a bandwidth limit of 1B per ingress packet under different transmission options.

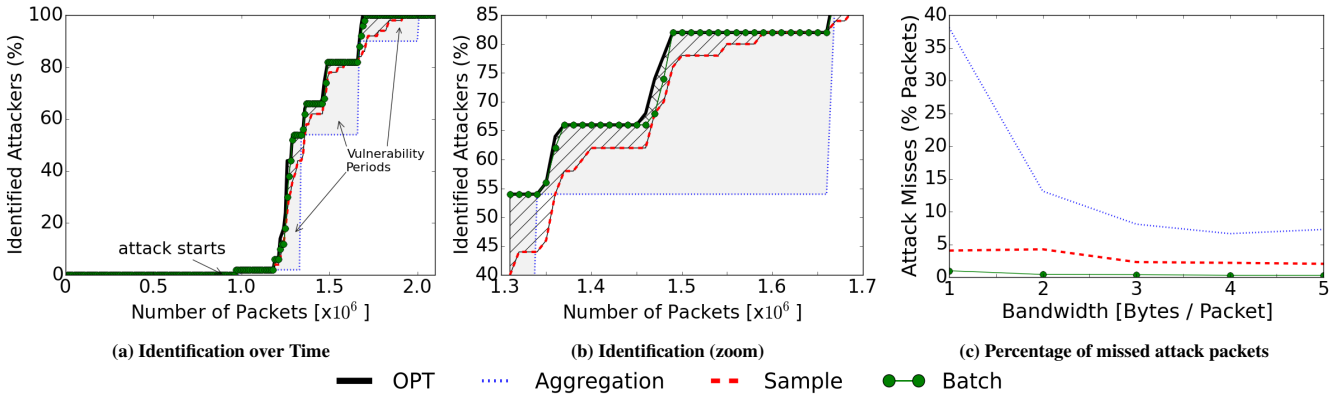


Figure 10: HTTP flood detection experiment. 50 attacking LANs on top of the backbone trace. Comparison of the detection time of the different approaches. Our batching approach achieves near-optimal detection time.

Traffic. We inject flood traffic on top of the Backbone packet trace. Specifically, we select a random time at which we inject 50 randomly-picked 8-bit subnets that account for 70% of the total traffic once the flood begins. We generate a new trace as follows. (1) We select 50 subnets by randomly choosing 8-bits for each, (2) and we select a random trace line in the range $(0,10^6)$. Until that line the trace is unmodified. (3) From that line on, at each line, with probability 0.7 we add a flood line from a uniformly picked flooding sub-network, and with probability 0.3 we skip to the next line of the original trace.

Results. Figure 10 depicts the results. Figure 10a and Figure 10b show the detection speed of the flooding subnets by the three different approaches at the controller. We compare among the three approaches and additionally outline an optimal algorithm that uses an accurate window and "knows" exactly what traffic enters the load-balancers without delay (OPT). It is notable that the Batch approach achieves near-optimal performance, and outperforms Sample and Aggregation. Figure 10c shows that the Batch method identifies almost all of the attack messages as is expected by our theoretical analysis. Further, its miss rate is 37× smaller under the 1-byte per packet bandwidth budget when compared to the ideal Aggregation method.

7 RELATED WORK

Heavy hitters are an active research area on both intervals [5, 10, 12, 13, 30, 38, 49] and sliding windows [6, 10, 11, 31, 54]. HH integration in the single-device mode is an active research challenge. For example, Sketchvisor [30] suggests using a lightweight fast-path measurement when the line is busy. This increases the throughput but reduces accuracy. Alternatively, HashPipe [49] adopts the interval-based Space Saving [38] into programmable switches. NetQRE [52] allows the network administrator to write a measurement program. The program can describe HH and HHH as well as sliding windows. However, their algorithm is exact rather than approximate and requires a space that is linear in the window size which is expensive for large windows.

Hierarchical heavy hitters. The HHH problem was first defined (in the Interval model) by [19], which also introduced the first algorithm. The problem then attracted a large body of follow-up work as well as an extension to multiple dimensions [8, 18, 20, 21, 27, 41, 53]. MST [41] is a conceptually simple multidimensional HHH algorithm that uses multiple independent HH instances; one instance is used for each prefix pattern. Upon a packet arrival, all instances are updated with their corresponding prefixes. The set of hierarchical heavy hitters is then calculated from the set of (plain) heavy hitters of each prefix type. The algorithm requires $O\left(\frac{H}{\epsilon}\right)$ space and $O(H)$ update time. MST can also operate in the sliding window model, by replacing the underlying HH algorithm with a sliding window solution [10, 11, 32]. Randomized HHH (RHH) [7] is similar to MST but only updates a single HH instance. This reduces the update complexity to a constant but requires a large amount of traffic to converge. RHHH does not naturally extend to sliding windows since each HH instance receives a slightly varying number of updates and thus considers a different window.

Network-wide measurement. The problem of network-wide measurement is becoming increasingly popular [26, 30, 37, 51]. A centralized controller collects data from all measurement points to form a network-wide perspective. Measurement points are placed in the network so that each packet is measured only once. The work of [3] suggests marking monitored packets which allows for more flexible measurement point placement.

In [26], the controller determines a dynamic reporting threshold that allows for reduced communication overheads. It is unclear how to utilize the method in the sliding window model. Yet, the optimization goal is very similar in essence to this work: maximize accuracy and minimize traffic overheads. Stroboscope [50] is another network-wide measurement system that also guarantees that the overheads adhere to a strict budget. FlowRadar [37] avoids communication during the entire measurement period. Instead, the state of each measurement point is shared at the end of the measurement. Thus, FlowRadar follows the Interval pattern, which we showed to be slow to detect new heavy hitters.

8 CONCLUSIONS

Our study highlights the potential benefits of sliding-window measurements to cloud operators and makes them practical for network

applications. Specifically, we showed that window-based measurements detect traffic changes faster, and thus enable more agile applications. Despite these benefits, sliding windows have not been used extensively, since existing window algorithms are too slow to cope with the line speed and do not provide a network-wide view. Accordingly, we introduced the Memento family of HH and HHH algorithms for both single-device and network-wide measurements. We analyzed the algorithms and extensively evaluated them on real traffic traces. Our evaluations indicate that the Memento algorithms meet the necessary speed and efficiently to provide network-wide visibility. Therefore, our work turns sliding-window HH and HHH measurements into a practical option for the next generation of network applications.

A potential drawback of existing HHH solutions, ours included, is the ability to make real-time queries. That is, while RHHH provides line-rate packet processing on streams and H-Memento provides it for sliding windows, neither allows sufficiently fast queries. Therefore, we believe that a mechanism that would allow constant-time updates for detection of changes in the hierarchical heavy hitters set would be a promising direction for future work.

We open-sourced the Memento algorithms and the HAProxy load-balancer extension that provides capabilities to block and rate-limit traffic from entire sub-networks (rather than from individual flows) [1]. We hope that our open-source code will further facilitate sliding-window measurements in network applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Kenjiro Cho, for their helpful comments and suggestions. This work was partly supported by the Hasso Plattner Institute Research School; the Zuckerman Institute; the Technion Hiroshi Fujiwara Cyber Security Research Center; and the Israel Cyber Bureau.

REFERENCES

[1] Memento algorithms code and HAProxy extension. https://github.com/DHMementoz/Memento.
[2] Unpublished, see http://www.lasr.cs.ucla.edu/ddos/traces/.
[3] Y. Afek, A. Bremler-Barr, S. L. Feibish, and L. Schiff. Detecting heavy flows in the SDN match and action model. *Computer Networks*, 136:1 – 12, 2018.
[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. *ACM SIGCOMM*, pages 435–446, 2013.
[5] D. Anderson, P. Bevan, K. Lang, E. Liberty, L. Rhodes, and J. Thaler. A high-performance algorithm for identifying frequent items in data streams. In *ACM Internet Measurement Conference*, pages 268–282, 2017.
[6] E. Assaf, R. Ben-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.
[7] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Constant time updates in hierarchical heavy hitters. In *ACM SIGCOMM*, 2017.
[8] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Volumetric hierarchical heavy hitters. In *IEEE MASCOTS*, 2018.
[9] R. B. Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargraftik, and E. Waisbard. Memento: Making sliding windows efficient for heavy hitters (full version). *CoRR*, 2018. http://arxiv.org/abs/1810.02899.
[10] R. Ben Basat, G. Einziger, and R. Friedman. Fast flow volume estimation. In *ICDCN '18*, 2018.
[11] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Heavy Hitters in Streams and Sliding Windows. In *IEEE Infocom*, 2016.
[12] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Optimal elephant flow detection. In *IEEE INFOCOM*, 2017.
[13] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *IEEE INFOCOM*, 2017.
[14] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild (univ 1 dataset). In *ACM Internet Measurement Conference*, 2010.
[15] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *ACM CoNEXT*, 2011.
[16] R. A. Brualdi. Introductory combinatorics. *New York*, 3, 1992.
[17] M. Chiesa, G. Rétvári, and M. Schapira. Lying your way to better traffic engineering. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ACM CoNEXT, 2016.
[18] K. Cho. Recursive lattice search: hierarchical heavy hitters revisited. In *ACM IMC*, 2017.
[19] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Data Streams. In *VLDB*, 2003.
[20] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-dimensional Data. In *SIGMOD*, pages 155–166, 2004.
[21] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Streaming Data. *ACM Trans. Knowl. Discov. Data*, 1(4):2:1–2:48, Feb. 2008.
[22] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 2002.
[23] G. Dittmann and A. Herkersdorf. Network processor load balancing for high-speed links. In *SPECTS*, volume 735, 2002.
[24] G. Einziger, M. C. Luizelli, and E. Waisbard. Constant time weighted frequency estimation for virtual network functionalities. In *ICCCN*, pages 1–9, July 2017.
[25] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *ACM SIGCOMM*, SIGCOMM '03, pages 137–148, New York, NY, USA, 2003. ACM.
[26] R. Harrison, Q. Cai, A. Gupta, and J. Rexford. Network-wide heavy hitter detection with commodity switches. In *ACM SOSR*, pages 8:1–8:7, 2018.
[27] J. Hershberger, N. Shrivastava, S. Suri, and C. D. Tóth. Space Complexity of Hierarchical Heavy Hitters in Multi-dimensional Data Streams. In *ACM PODS*, pages 338–347, 2005.
[28] P. Hick. CAIDA Anonymized 2016 Internet Trace, equinix-chicago 2016-02-18 13:00-13:05 UTC, Direction A.
[29] S. Hilton. Dyn Analysis Summary Of Friday October 21 Attack. Available: https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.
[30] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, 2017.
[31] R. Y. S. Hung, L. Lee, and H. Ting. Finding frequent items over sliding windows with constant update time. *IPL 2010*.
[32] R. Y. S. Hung and H. F. Ting. Finding heavy hitters over the sliding window of a weighted data stream. In *LATIN*, pages 699–710, 2008.
[33] L. Jose, M. Yu, and J. Rexford. Online measurement of large traffic aggregates on commodity switches. In *USENIX Hot-ICE*, 2011.
[34] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford. Clove: Congestion-aware load-balancing at the virtual edge. In *ACM CoNEXT*, 2017.
[35] A. Khalimonenko, O. Kupreev, and E. Badovskaya. DDoS attacks in Q1 2018. Available: https://securelist.com/ddos-report-in-q1-2018/85373/.
[36] L. K. Lee and H. F. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *ACM PODS*, 2006.
[37] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *Usenix NSDI*, 2016.
[38] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*, 2005.
[39] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, pages 15–28, 2017.
[40] M. Mitzenmacher, T. Steinke, and J. Thaler. Hierarchical heavy hitters with the space saving algorithm. *CoRR*, 2011. Conference version appeared in ALENEX 2012.
[41] M. Mitzenmacher, T. Steinke, and J. Thaler. Hierarchical Heavy Hitters with the Space Saving Algorithm. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, ALENEX, 2012.
[42] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, 2014.
[43] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 2005.
[44] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1, 2005.
[45] K. Nyalkalkar, S. Sinhay, M. Bailey, and F. Jahanian. A comparative study of two network-based anomaly detection methods. In *IEEE Infocom*, 2011.
[46] T. Peng, C. Leckie, and K. Ramamohanarao. Protection from distributed denial of service attack using history-based ip filtering. 2002.
[47] R. 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 Conference on Internet Measurement*, IMC '04, pages 207–212, New York, NY, USA, 2004. ACM.

[48] V. Sekar, N. G. Duffield, O. Spatscheck, J. E. van der Merwe, and H. Zhang. Lads: Large-scale automated ddos detection system. In *USENIX ATC*, 2006.

[49] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR*, 2017.

[50] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever. Stroboscope: Declarative network monitoring on a budget. In *Usenix NSDI*, 2018.

[51] T. Yang, J. Jiang, P. Liu, J. G. Qun Huang, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. *ACM SIGCOMM*, 2018.

[52] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. Quantitative network monitoring with NetQRE. In *ACM SIGCOMM*, pages 99–112, 2017.

[53] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications. In *ACM IMC*, pages 101–114, 2004.

[54] Y. Zhou, Y. Zhou, S. Chen, and Y. Zhang. Per-flow counting for big network data stream over sliding windows. In *IEEE IWQoS*, 2017.

[55] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. *ACM SIGCOMM CCR*, 45(4), Aug. 2015.