Cuckoo Sampling: Robust Collection of Flow Aggregates under a Fixed Memory Budget

Josep Sanjuàs-Cuxart* Pere Barlet-Ros* Nick Duffield‡ Ramana Kompella†
*UPC BarcelonaTech ‡AT&T Labs–Research †Purdue University

***Abstract*—Collecting per-flow aggregates in high-speed links is challenging and usually requires traffic sampling to handle peak rates and extreme traffic mixes. Static selection of sampling rates is problematic, since worst-case resource usage is orders of magnitude higher than the average. To address this issue, adaptive schemes have been proposed in the last few years that periodically adjust packet sampling rates to network conditions. However, such proposals rely on complex algorithms and data structures of costly maintenance. As a consequence, adaptive sampling is still not widely implemented in routers.**

We present a novel flow sampling based measurement scheme called Cuckoo Sampling that efficiently collects per-flow aggregates, while smoothly discarding information as it exceeds the available memory. After a measurement epoch, it provides a random sample of the input flows, at a close-to-maximum rate as allowed by the available memory budget.

Our proposal relies on a very simple data structure, requires few per-packet operations, has a CPU cost that is independent of the memory budget and traffic profile, and is suitable for hardware implementation. We back the theoretical analysis of the algorithm with experiments with synthetic network traffic, and show that our algorithm requires significantly less resources than existing adaptive sampling schemes.

I. Introduction

As networks grow more complex and hard to manage, the deployment of devices that monitor network conditions has become a necessity. Network monitoring can aid in tasks such as fault diagnosis and troubleshooting, evaluation of network performance, capacity planning, traffic accounting and classification, and to detect anomalies and investigate security incidents. However, network traffic analysis is challenging in high-speed data links. In current backbone links, incoming packet rates leave very little time (e.g., 32 ns in OC-192 links in the worst case) to process each packet. Additionally, storing all traffic is inviable; usually, operators only record traffic aggregates on a per-flow basis, as a means to obtain significant data volume reduction.

A paradigmatic example and, arguably, the most widespread flow-level measurement tool is NetFlow [1], which provides routers with the ability to export per-flow traffic aggregates. However, in today's networks, one can expect the number of active flows to be very large and highly volatile. Under anomalous conditions, including network attacks such as worm outbreaks, network scans, or even attacks that target the measurement infrastructure itself, the number of active flows can rise by orders of magnitude. Thus, not only must the router be able to process each packet very quickly, but must also maintain a potentially enormous amount of state. As a consequence, provisioning monitors for worst-case scenarios is prohibitively expensive [2].

The most widely adopted approach both to prevent memory exhaustion and to reduce packet processing time is to sample the traffic under analysis. For example, Sampled NetFlow [1] is a standard mechanism that samples the incoming traffic on a per-packet basis. Sampled NetFlow requires the configuration of a fixed (static) sampling rate by the network operator. The main problem of such an approach is that operators tend to select "safe" parameters that ensure network devices will continue to operate under adverse traffic conditions. As a result, the sampling rates are set with the worst-case scenario in mind, which harms the completeness of the measurements under normal conditions.

Several works have addressed the problem of dynamic packet sampling rate selection, which overcomes the drawbacks of setting static sampling rates by adapting to network conditions (e.g., [2]–[4]). Most notably, Adaptive NetFlow (ANF) [3] maintains a table of active flows; when it fills, it lowers the sampling rate and updates all table entries as though packets had been initially sampled at the resulting (lower) rate; flows for which the packet count becomes zero are discarded.

However, adaptive sampling schemes, including ANF, are still not widely used. For example, NetFlow still relies on static sampling. We believe that the main reasons for this are that existing adaptive sampling schemes are too costly in terms of CPU requirements, and rely on complex data structures and algorithms, which makes them less attractive for implementation in networking hardware (we review the related work in Sec. II, while Sec. III presents ANF in greater detail).

In this work, we turn our attention to flow-wise packet sampling [5] (also known as flow sampling), which allows us to find an elegant solution to the problem of adaptive sampling. We present a novel measurement scheme which we have named *Cuckoo Sampling* (Sec. IV) that performs aggregate per-flow network measurements and, when the state required to track all incoming traffic exceeds a memory budget, maintains the largest possible random selection of the incoming flows, i.e., *under overload, performs flow sampling at the appropriate rate*. Our algorithm can cope with the extreme data rates of today's fast network links. The data structure is extremely efficient both when the traffic conforms to the available memory budget, but also under overload, when flow sampling is necessary. We provide analytic (Sec V) and experimental (Sec. VI) evidence of the efficiency of our proposal.

978-1-4673-0775-8/12/$26.00 ©2012 IEEE

II. Related Work

A classical solution to reduce the load of network monitors is traffic sampling. Several sampling methods have been proposed in the literature; a review of the most relevant can be found in [5]. Each sampling method preserves certain characteristics of the input traffic; sampling mechanisms should be considered to be complementary. An alternative approach to traffic sampling is sketching (e.g., [6]); a comparison between sketching and sampling can be found in [7].

Perhaps the simplest and most widely used is uniform random packet sampling (PS). Flow-wise packet sampling [5], also known as flow sampling (FS), requires each data flow to be either completely sampled or discarded, which can be effortlessly done using hash-based sampling. PS biases collection towards large flows, and makes it very hard to recover per-flow properties, such as the flow size distribution [8]. On the other hand, FS preserves flow aggregates, but is less accurate for applications such as volume based traffic accounting, or for heavy hitter detection, due to the heavy-tailed nature of flow size distributions [9]. Several studies analyze the impact of sampling on the accuracy of other monitoring applications, e.g., flow accounting [10] and anomaly detection [11]–[13].

As explained, statically setting sampling rates is problematic. An alternative solution is to adapt sampling rates according to traffic conditions. The most relevant related work to our solution is Adaptive NetFlow [3]. Given its relevance to our work, we devote Sec. III to introduce this proposal in detail. Flow Slicing [4] is a recently appeared technique that combines Sample and Hold [6] and PS in a way that can simultaneously control memory usage and the volume of output results. The problem of adaptively choosing sampling rates in a system running multiple monitoring applications has been investigated in [2]. The abstract problem of sampling a pre-defined number of items from a set is called Reservoir Sampling [14]. The algorithm we present is reminiscent of Cuckoo Hashing [15] (CH); hence, we have named our technique Cuckoo Sampling (CS).

III. Background

Adaptive NetFlow (ANF) is a proposal to implement adaptive packet sampling in routers that export traffic information via NetFlow records. ANF initially samples all packets, and starts collecting flow aggregates in a table. The core idea behind ANF is that, when the memory becomes full, the packet sampling rate for future packets is lowered and, simultaneously, all existing flow entries are modified as though they had been initially sampled at the resulting rate, a procedure that is known as *renormalization*.

The objective of renormalization is to delete flow entries for which packet counts reach zero, thus freeing space for new entries. This process is repeated as necessary as new flows arrive, and runs in parallel with regular packet processing. Naive renormalization would involve a costly binomial random number generation. ANF achieves similar results with a single coin flip per stored flow.

The choice of the new sampling rate is critical since, for a given sampling rate, the fraction of flows that will be discarded depends on the distribution of the traffic. For this reason, ANF also maintains a histogram of flow sizes. Given a target fraction of flows to discard f, and the flow size distribution, the new sampling rate can be computed. This implies that (e.g., if traffic measurement is about to end) the algorithm might unnecessarily discard up to f of its samples. Of course, this problem could be mitigated by provisioning the monitor with additional memory. We argue, though, that flow aggregate collection data structures must not only require a small number of memory accesses per packet, but should also be memory efficient to allow line-speed monitoring with fast (and therefore, more expensive) memory modules.

IV. Cuckoo Sampling

Measurement of a single flow. Let us start with the assumption that we have memory for exactly one flow. We can easily sample one random flow by using hash-based sampling, with the intention of storing the flow with lowest hash. For every incoming packet, a pseudo-random hash of its flow identifier is calculated and compared against that of the currently stored flow. If smaller, discard the old flow, and store the new one. If larger, ignore the packet. If flow identifiers match, simply update flow information.

Arbitrary memory budget. The algorithm just outlined could be extended to measure b flows as follows. Maintain as many instances of the previous scheme as the memory budget allows. Then, use the hash function h on the flow identifier id to randomly pick an instance $i = h(id) \mod b$. Each instance then independently runs the algorithm previously described.

This approach is memory effective and CPU efficient, but wastes a significant amount of memory. In particular, right when the number of flows matches the memory budget, i.e., *the monitor is correctly dimensioned*, $e^{-1} \approx 36.8\%$ of the memory remains unused (see Sec. V).

The complete algorithm. Cuckoo Hashing solves the issue of memory under-utilization by having each flow hash to multiple locations. Our data structure is composed of an array of b buckets and $k+1$ pseudo-random hash functions. Each bucket contains a flow hash, and the attached flow information, possibly including its identifier and aggregate statistics, such as those provided by NetFlow. Per-packet operations are as follows (the full algorithm can be observed in Figure 1).

When a packet arrives, its flow identifier id is hashed by $k+1$ pseudo-random hash functions. Functions $h_1..h_k$ have range $[0, b-1]$, and determine k positions in the array. Additionally, function h determines what we call *the flow's hash value*, which is stored in the bucket where the flow resides.

The k positions in the array are verified. If a matching flow identifier is found, its entry is updated. Otherwise, the first empty position, if any, is used. When a new entry is created, the flow's hash $h(id)$ is recorded in the flow's bucket.

When none of the k positions are empty, the algorithm checks which of the k positions holds *the largest hash*. Let that position be L. Then, it proceeds to compare the hash stored

```
1: function InsertAggregate(flow, value)
2:     for i=1, K do
3:         p ← h_i(flow); info ← table[p]
4:         if info undefined then                    ▷ found empty spot
5:             table[p] ← ⟨h(flow), flow, value⟩; return
6:         end if
7:         if info.hash = h(flow) and info.flow = flow then
8:             table[p].value += value; return       ▷ update flow info
9:         end if
10:        if i = 1 or info.hash > max.hash then
11:            max ← info; maxp ← p                  ▷ track largest hash
12:        end if
13:    end for
14:    if h(flow) ≥ max.hash then
15:        return                                    ▷ large flow hash, discard
16:    end if
17:    table[maxp] ← ⟨h(flow), flow, value⟩          ▷ insert new
18:    InsertAggregate(max.flow, max.value)          ▷ relocate old
19: end function
```

Fig. 1. Cuckoo Sampling algorithm.

in bucket b_L against $h(id)$. If it is smaller, it means that all of the k positions hold smaller hashes. Hence, the current packet is discarded, and its flow will never have the chance to enter the data structure, as will be discussed later.

Otherwise, the packet will enter the data structure, and take position L. However, the flow stored in this position are not simply discarded. Instead, we attempt to re-locate the flow recursively. That is, again, we determine its k possible positions, and repeat the previous scheme. Note that this might, in turn, trigger additional relocation of other flow entries (we will show that the number of necessary relocations is very small in Sec. V). The main advantage of this scheme is that it greatly increases worst-case memory usage (see Sec.V).

An alert reader might question the need to recursively relocate flows. Why not just discard the older flow instead? The recursive relocation procedure guarantees that, when a flow is discarded, it will never be considered for re-inclusion in the data structure. Let position p, occupied by a flow with identifier id, have been claimed by a flow with smaller hash value. The replaced flow hashed to positions $P = \{h_1(id), .., h_k(id)\}$, with $p \in P$. Then, consider the case that $h(id)$ is not the largest among the hashes stored in positions P. When a new packet of flow id arrived, the algorithm would determine positions P. Since $h(id)$, as we previously assumed, is not the highest among P, the algorithm would determine that the packet should enter the data structure by replacing the now worst hash of P. Therefore, a new entry for flow id would be created. However, this entry would not aggregate all the packets of id, which clearly violates the objective of either sampling or discarding *entire* flows.

V. Analysis and Parametrization

A. Memory Efficiency

We start by analyzing the memory efficiency of the algorithm with b buckets, $k = 1$, i.e., with a single hash function, and n incoming flows. The probability that a given bucket is never hit by a flow is $(1 - 1/b)^n \approx e^{-n/b}$.

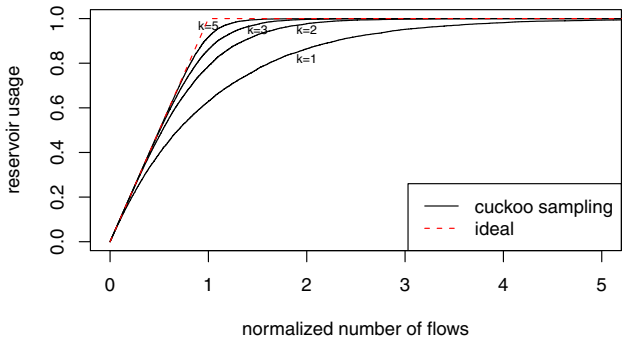

Fig. 2. Reservoir usage obtained using cuckoo sampling with different number of hash functions.

Thus, the expectation of the number of measured flows is $E[m] = b(1 - e^{-n/b})$. Of course, the data structure can only measure up to b flows. We can define the memory efficiency ratio of the data structure as $m/min(n, b)$. This expression is minimized when $n = b$, which renders e^{-1} buckets unused.

Additional hash functions mitigate this worst case. We provide a lower bound on the expectation of the number of used buckets from a simplified model of the algorithm, under which, if all k hashes of a previously unseen flow key map to a occupied location, the flow is simply discarded. Suppose n distinct keys have arrived, and let $X_n \leq n$ be the number of these that are stored. Then $\{X_n : n \geq 1\}$ is a pure birth process indexed by "time" n, with $X_1 = 1$, and transitions:

$$X_n \mapsto X_{n+1} = \begin{cases} 1 + X_n, & \text{with probability } 1 - p(X_n) \\ X_n, & \text{with probability } p(X_n) \end{cases}$$

where $p(x) = (x/b)^k$ is the probability that all k hash functions maps to an occupied location. The "lifetime" in each level x of X, i.e., the number of additional unseen keys before X increments, is then geometrically distributed $1/(1 - p(x))$. Thus the average total "time" taken to get to a level z (i.e. the average number of distinct keys that result in z slots being occupied) is $T(z) = \sum_{x=0}^{z-1} 1/(1 - p(x))$. The benefit introduced by each additional hash function diminishes, as can be observed in Figure 2, which presents the percentage of occupied buckets (including relocations) as a function of the number incoming of flows. (We normalize the number of flows by dividing over the reservoir size.)

B. Cost

Collecting traffic aggregates is computationally inexpensive; the bulk of the cost comes from managing the data structures and performing the necessary memory accesses. Hence, we measure the CPU cost of our algorithm in terms of memory accesses, which we analyze in this subsection.

When it runs out of empty buckets, our algorithm starts to recursively relocate items in the array of buckets. How large is this cost? Let us consider an array that is fully populated, i.e., it contains no unused buckets. This is clearly a worst case, since it presents less opportunities to cut the chain of successive relocations.

We consider the algorithm starts at recursivity level 1, and wish to calculate the expectation of the number of recurses that the algorithm will perform. Let P_i be the probability of advancing past level i, once it has been reached. In recursivity levels $i \geq 2$, the algorithm has already visited $x_i = i\,(k-1)+1$ buckets in the previous levels. In the current level, it is trying to relocate the largest hash it has encountered so far into one of the new $k-1$ positions (one position is shared with the previous level). Another relocation will be triggered if, in the new level, an even larger hash is found, which happens with probability $P_i = \frac{k-1}{x_i+k-1}$ for $i \geq 2$. Let us assume an also pessimistic $P_1 = k/(k+1)$, which corresponds to the case where the array is populated with random hash values that have never been replaced; P_1 can only be smaller in practice.

Then, the probability that the algorithm performs exactly i recurses is $R_i = (1-P_i)\prod_{j=1}^{i-1} P_j$, and the expectation for the number of recurses is $\sum_{i=1}^{\infty} iR_i$. It can be shown that each of the iR_i terms is smaller than $1/i!$ and, thus, the sum is smaller than e. Therefore, we can conclude that the expected number of relocations per flow arrival is a constant smaller than e.

Every recurse requires, at most, k memory accesses. Hence, expected per-packet cost is no worse than $e\,k$ accesses, even in worst-case scenarios where each packet belongs to a new flow (i.e., a DoS attack such as a SYN flood [16]). Since small values of k are already practical, k can also be considered constant. Therefore, our algorithm's processing cost is linear to the number of packets. Note also that this bound on the per-packet algorithm's cost has the remarkable property that it does not depend on the reservoir size. This feature clearly differentiates this measurement scheme from alternative approaches and, especially, Adaptive NetFlow. As for memory usage, our algorithm is linear to the reservoir size.

We end by noting that, as revealed by the analysis, the probability of entering successive recursivity levels greatly diminishes. This means that, in an implementation of the technique, the number of relocations can be artificially cut, while incurring an arbitrarily small risk of damaging the measurements. The probability of performing i or more successive relocations is below $\sum_{j=i}^{\infty} 1/j!$, e.g., for 9 relocations, below $\approx 3 \times 10^{-6}$. This is desirable for ease of implementation with a hardware pipeline.

VI. Simulation Results

In this section we analyze the methods under synthetic traffic by generating a large number of 1-packet flows, which is a worst-case scenario for collecting traffic aggregates. Generally, successive packet arrivals are not overly interesting, since the sampling decision has already been taken, and have smaller cost. Additionally, this scenario mimics an extreme DoS attack, and puts the measurement algorithms under maximal stress. (We also performed experiments with real traffic that confirmed the results presented in this section, but were unable to include them in the interest of space.) We set an example configuration with reservoir size 10000, and generate 5 times as many 1-packet flows.

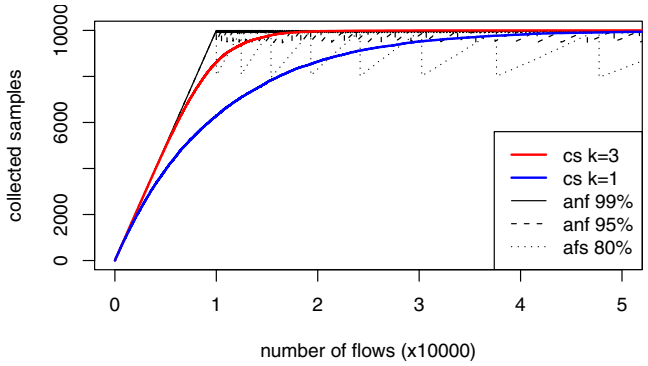

Fig. 3. Sample size obtained with several configurations.

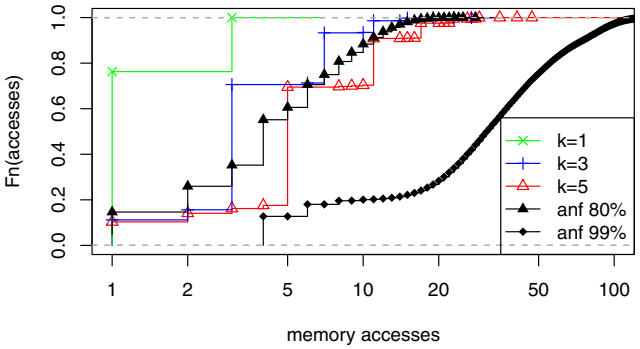

Fig. 4. CDF of the number of memory accesses with several values of k, reservoir of size 10^4 and 5 times as many flow arrivals.

We have tested several configurations of Cuckoo Sampling (CS) with k hash functions (referred to as CS-k), and Adaptive NetFlow (ANF) with different α values (ANF-α), with $1-\alpha$ corresponding to the fraction of flows that ANF evicts in every renormalization (ANF is thoroughly described in Sec. III). Our implementation of ANF assumes perfect knowledge of the flow size distribution, i.e., we do not implement nor count the necessary memory access to maintain a histogram of flow sizes. As for the flow table, we use a standard hash table with as many buckets as the reservoir size.

Figure 3 shows the sample size we obtained. CS-3 achieves higher than ANF-80% worst-case memory usage (on the 10^4th flow), but clearly outperforms it as more flows arrive. The figure includes CS-1 as a reference. Note that CS-3 performs as well as ANF-99% after 2×10^4 flow arrivals and, as will be shown later, has much lower CPU cost. For clarity of presentation, this figure excludes CS-5, but its behavior can be predicted from Figure 2.

We next analyze the number of memory accesses that each configuration requires. Figure 4 presents the CDF of the number of memory accesses of a few reference configurations. CS-3 outperforms ANF-80%, while having similar worst-case memory usage.

However, the CDFs fail to capture an important particularity of ANF, which is that its cost spikes when the maintenance procedure starts. Figure 5 shows the average memory accesses of incoming packets, binned in groups of 1000. The cost

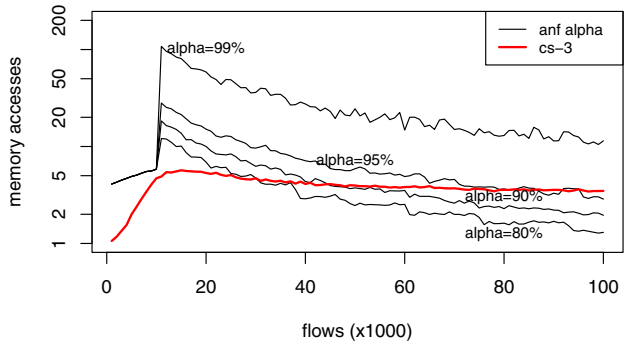

Fig. 5. Average cost per flow in bins of 1000 flows.

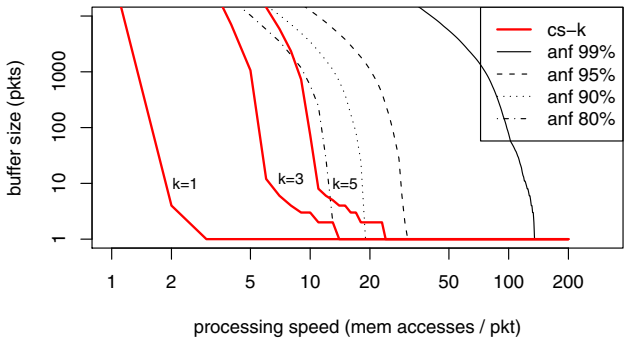

Fig. 6. Buffer size required to sample 10^4 flows as a function of the processing speed.

of ANF decreases in the long term. We argue that this is irrelevant, since a network monitor will require either the processing power to absorb the peak, or a large buffer capable of absorbing incoming packets while others are being processed. In contrast, CS does not present spikes, so it will demand less from the CPU, and will require almost no buffering. Note the logarithmic axis: ANF-99% peaks at 107; ANF-95% at 28, while CS-3 is around 5.

Figure 6 analyzes the required necessary buffer to absorb any cost peaks, according to the processing capabilities of the monitor, expressed in number of memory accesses that can be performed per packet arrival. To provide a fair basis for comparison, we have implemented the normalization process of ANF as a background process, i.e., packets do not need to wait until normalization is complete to enter the data structure. The figure shows that CS is less resource demanding. For example, CS-5 severely outperforms ANF-99%, since it only buffers 8 packets using 11 accesses/pkt, whereas ANF-99% requires 124 accesses/pkt to achieve the same buffer usage (note the logarithmic axes). As discussed in Sec. V-B, the number of relocations can be safely capped, which would reduce buffer usage for CS even further.

VII. Conclusions

Dynamically adjusting sampling rates is crucial to extract as much information as possible from the input traffic, without exceeding the monitor's resources. The literature provided a packet-sampling based methods (most notably, Adaptive NetFlow) that followed this approach. However, adaptive packet sampling schemes have not been widely implemented, possibly due to their complexity in terms of both implementation and hardware requirements.

In this paper, we turned our attention to flow-wise packet sampling, and presented a novel technique called Cuckoo Sampling that performs adaptive flow sampling. We propose a simple randomized data structure that has very small (constant) per-packet cost and is very easy to parametrize. Compared to previous approaches, it is based on an extremely simpler algorithm that can be expressed in a few lines of pseudocode and, most importantly, requires fewer hardware resources than adaptive packet sampling. A very notable feature of this algorithm is that its per-packet cost is independent of the size of the flow store.

Additionally, we have shown that it is suitable for implementation with a hardware pipeline. We have analyzed the method using both synthetic and real traffic to verify that the method behaves as predicted by the theoretical analysis. Our experiments included an extremely challenging traffic profile formed of 1-packet flows that mimicked a distributed denial of service attack. As a conclusion of this work, we believe this method to be very practical and, in particular, to be suitable for implementation within routers.

Acknowledgments

This research was funded by the Spanish Ministry of Science and Innovation under contract TEC2011-27474 (NOMADS) and *Comissionat per a Universitats i Recerca del DIUE de la Generalitat de Catalunya* (ref. 2009SGR-1140).

References

[1] Cisco, "NetFlow," http://www.cisco.com/web/go/netflow.
[2] P. Barlet-Ros, G. Iannaccone *et al.*, "Load shedding in network monitoring applications," in *Proc. of USENIX ATC*, 2007.
[3] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," in *Proc. of ACM SIGCOMM*, 2004.
[4] R. Kompella and C. Estan, "The power of slicing in internet flow measurement," in *Proc. of ACM/USENIX IMC*, 2005.
[5] N. Duffield, "Sampling for Passive Internet Measurement: A Review," *Statistical Science*, vol. 19, no. 3, pp. 472–498, Aug. 2004.
[6] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 270–313, 2003.
[7] P. Tune and D. Veitch, "Sampling vs sketching: An information theoretic comparison," in *Proc. of IEEE INFOCOM*, 2011.
[8] N. Hohn and D. Veitch, "Inverting sampled traffic," in *Proc. of ACM/USENIX IMC*, 2003.
[9] N. Duffield, C. Lund, and M. Thorup, "Charging from sampled network usage," in *Proc. of ACM SIGCOMM IMW*, 2001.
[10] T. Zseby, T. Hirsch, and B. Claise, "Packet sampling for flow accounting: Challenges and limitations," in *Proc. of PAM*, 2008.
[11] J. Mai, A. Sridharan, C. Chuah, H. Zang, and T. Ye, "Impact of packet sampling on portscan detection," *IEEE JSAC*, vol. 24, no. 12, pp. 2285–2298, Dec. 2006.
[12] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye *et al.*, "Is sampled data sufficient for anomaly detection?" in *Proc. of ACM/USENIX IMC*, 2006.
[13] D. Brauckhoff, B. Tellenbach *et al.*, "Impact of packet sampling on anomaly detection metrics," in *Proc. of ACM/USENIX IMC*, 2006.
[14] J. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985.
[15] R. Pagh, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.
[16] J. Lemon *et al.*, "Resisting syn flood dos attacks with a syn cache," in *Proc. of BSDCon*, 2002.