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 both synthetic and real 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 [3] maintains a table of active flows; when the table becomes full, the algorithm 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 Adaptive NetFlow, are still not widely used. For example, Cisco's 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 Adaptive NetFlow 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 and Sec. VII) evidence of the efficiency of our proposal. An important finding of this work is that Cuckoo Sampling is both easier to implement and less resource demanding than adaptive packet sampling. Our scheme has smaller peak costs, can smoothly discard excess flows, and relies on very simple data structures and algorithms. Unlike Adaptive NetFlow and other alternative data structures, it exhibits an expected constant per-packet cost, i.e., its cost is independent of the memory budget. Additionally, our algorithm is very easy to parametrize, and is independent of the traffic profile, especially in front of attacks or aggressive traffic patterns. For all these reasons, we believe it to be more practical for hardware implementation within routers.

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]. Perhaps the simplest and most widely used is uniform random packet sampling, which is easy to implement by generating a random value for each packet arrival, and can guarantee a reduction in the per-packet processing cost. However, it does not proportionally reduce the amount of memory when computing per-flow aggregates [6], which forces operators to set low sampling rates.

Trajectory sampling [7] provides a means to coordinate the sampling across several monitors along a packet path, using a pre-arranged pseudo-random hash function. Sampling decisions are based on the hash values of each packet; all monitors use the same hash function, so that packet selection is coordinated.

Flow-wise packet sampling [5], also known as flow sampling, requires each data flow to be either completely sampled or discarded. Achieving this effect might look challenging, but can also be effortlessly done using hash-based sampling. For every packet, a hash of the flow identifier is computed. If this value is below a certain threshold, the flow is sampled.

Each sampling method preserves certain characteristics of the input traffic. Sampling mechanisms should be considered to be complementary [5]. Packet sampling biases data collection towards large flows, and makes it very hard to recover per-flow statistical properties, such as the original flow size distribution [8]. On the other hand, flow sampling 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].

Sampling methods have been proposed for specific traffic metrics, such as the number and average length of flows [14] or flow size distribution [15], and to detect flows of particular interest [16]–[18]. Another family of sampling algorithms operate after packets have been aggregated by routers. Ref. [19] proposes a sampling method to select, under hard memory constraints, a representative subset of the NetFlow records exported by a router, but this solution is not applicable for traffic sampling within routers themselves.

Statically setting sampling rates is problematic, since they are usually selected with worst-case traffic conditions in mind. 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 [16] and packet sampling 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 [20]. Initially, all elements are sampled, until the reservoir is fully populated. Then, every new element replaces a randomly chosen element of the reservoir with a certain probability, so that every item is equally likely to be selected. Our technique can be seen as an efficient design of Reservoir Sampling to collect per-flow aggregates.

The algorithm we present stores items in an array and, as packets arrive, can relocate items to alternative positions within the array. This is reminiscent of the way the Cuckoo Hashing [21] data structure operates; hence, we have named our technique Cuckoo Sampling. Note however that this is merely anecdotal, since both data structures operate very differently.

III. Background

As has been explained, statically setting sampling rates is harmful for measurement accuracy, since it forces operators to choose sampling rates with worst-case scenarios in mind. A more sensible solution is to adapt sampling rates to traffic conditions. Adaptive NetFlow (ANF) is a proposal to implement adaptive packet sampling in routers that export traffic information via NetFlow records. Given its close relationship with our work, we devote this section to introduce it in more detail.

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 binomial random number generation, which is costly. ANF tackles this issue by proposing a method that 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

In this section, we present an extremely simple data structure that is capable of performing adaptive flow sampling with a given memory budget. The main novelty of our proposal is the algorithm to update the data structure, which is also very simple, and requires considerably lower cost than Adaptive NetFlow, as will be discussed in Secs. VI and VII.

A. Measurement of a Single Flow

Let us start with the assumption that we have memory for exactly one flow. Thus, we wish to devise an algorithm that randomly selects one flow from the traffic, i.e., operates with a reservoir of size 1. Under such a setting, reservoir sampling takes new items with probability $1/n$, where n corresponds to the number of arrivals so far. However, this scheme would require tracking active flows (e.g., in a hash table) in order to discern if a packet belongs to a new flow that can replace the existing one.

We can easily obtain a similar result without storing per-flow state 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 it is smaller, the existing flow is discarded in favor of the newer. If it is larger, the packet is discarded. Finally, if it matches, the flow information is updated. As long as the hash function is perfectly pseudo-random, and has a large enough range, it is easy to see that the algorithm will select a flow randomly, given that all flows have exactly the same probability of obtaining the lowest value.

B. Arbitrary Reservoir Size

As explained, devising an algorithm that randomly selects one flow from the traffic is simple and computationally lightweight. However, network operators wish to obtain as much data as possible, according to the memory budget of the measurement device. Under normal operating conditions, this budget will be sufficient to track most active flows. However, the data structure must be robust to (possibly, sudden) increases in the number of flows well beyond the available memory budget. It is in such a scenario where we wish the data structure to perform random flow sampling at the appropriate rate.

We propose an initial step towards that goal by trying to apply the basic algorithm outlined in Sec. IV-A and extending it to multiple flow measurement. One could hope to extend it straightforwardly to a reservoir of size r 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 $i = h(id) \mod r$. Each instance then runs the algorithm described in Sec. IV-A independently.

This approach presents important advantages. Firstly, it is based on a very simple data structure. Traditional hash tables or other kind of dictionaries, including those based on Cuckoo Hashing [21], require collision management. Secondly, its per-packet cost is constant, unlike those reservoir sampling algorithms that rely on maintaining secondary data structures to perform flow selection (e.g., a heap), where cost depends on the reservoir size. Third, unlike Adaptive NetFlow, it does not require periodic maintenance to renormalize entries, which would cause spikes in the overall cost.

Despite these advantages, this scheme is not effective because, when operating under normal conditions (i.e., no overload), it wastes a significant amount of memory. In particular, when the number of incoming flows is in the order of the available memory budget, i.e., the memory is well dimensioned to handle incoming flows, around $e^{-1} \approx 36.8\%$ of the memory remains unused due to collisions, as will be further discussed in Sec. V. This approach would therefore require provisioning an additional $\approx 60\%$ of memory than would initially seem necessary.

C. The Complete Algorithm

In this subsection, we present the complete data structure and packet processing algorithm that form the core of our proposal. As explained, the main problem of the scheme outlined in the previous subsection is that a large amount of memory is wasted due to collisions precisely when the monitor is correctly dimensioned.

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 the flow identifier and aggregate statistics, such as the total number of packets or bytes, or any number of NetFlow equivalent fields. The 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 of counters. Additionally, hash function h determines what we call *the flow's hash value*, which must always be stored in the bucket where the flow will reside.

The k positions in the array of counters are verified. If a matching flow identifier is found, the corresponding entry is updated. Otherwise, the first empty position, if any, is used.

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

Fig. 1. Cuckoo Sampling algorithm.

When a new entry is created, the flow's hash $h(id)$ is recorded in the flow's bucket.

As more flows enter the data structure, the data structure starts facing overload, and the probability of finding an unused position decreases. When none of the k positions are empty, the algorithm performs the following procedure. First, it checks which of the k positions holds the largest hash. Let that position be L. Then, it proceeds to compare the hash stored in bucket b_L against $h(id)$.

If the stored hash is smaller than the current flow's, it means that the k positions where the current flow hashes all hold smaller hashes. Analogously to the scheme presented in Sec. IV-A, the current packet is discarded, and its flow will never have the chance to enter the data structure, as will be discussed later.

If it is not smaller, the packet will enter the data structure, and take position L. However, the flow stored in this position can not be simply discarded, as will be explained in Sec. IV-D. 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, since it provides greater opportunities for flows to occupy unused memory positions, compared to the previous scheme described in Sec. IV-B, as will be shown in Sec.V.

D. The Flow Sampling Guarantee

An alert reader might question the need to recursively relocate flows when another one with a lower hash value enters the data structure. Why can the data structure not simply discard the older flow? Let position p, occupied by a flow with identifier id, have been claimed by a flow that has a smaller hash value. The replaced flow hashed to the set of positions $P = \{h_1(id), .., h_k(id)\}$, with $p \in P$. Then, consider the case that $h(id)$ is not the largest of the hashes stored in positions P, which, incidentally, is a common case. What would happen if the algorithm just discarded the existing flow, instead of relocating it?

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.

In other words, the recursive relocation procedure guarantees that, when a flow is discarded, it will never be considered for re-inclusion in the data structure. Additionally, since the decision to include or reject flows is based on pseudo-random hashes of flow IDs, the selection of flows is also pseudo-random. Therefore, under overload, the data structure performs random flow sampling.

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 $(1 - 1/b)^n \approx e^{-n/b}$. 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. That is, around 37% of the memory is wasted.

By introducing additional hash functions, this worst case is mitigated. Every additional hash function provides opportunities for flows to occupy alternative buckets, in case of collision with a flow with lower hash. We can provide a lower bound on the expectation of the number of used buckets from a simplified model of the algorithm.

In this model, if all k hashes of a previously unseen flow key map to a occupied location, the flow is immediately discarded without attempt at relocation (or, equivalently, the algorithm does not attempt to relocate evicted flows). It is easy to see that the actual algorithm can only obtain equal or higher memory usage.

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

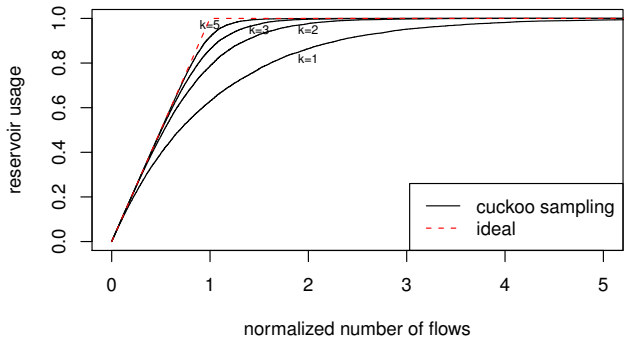

Fig. 2. Reservoir usage obtained using cuckoo sampling with different number of hash functions.

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:

$$T(z) = \sum_{x=0}^{z-1} 1/(1-p(x)).$$

In this approximation, we can then exhibit the store utilization as function of the number of distinct keys by plotting points with coordinates $(T(z), z/b)$ for $z = 1, 2, ..., b$.

The benefit introduced by each additional hash function diminishes, as can be observed in Figure 2. The figure presents the percentage of occupied buckets including relocations, as a function of the number incoming of flows. To avoid tying the figure to any particular case, 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: if the array has empty positions, it presents more 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 recursivity 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.

We are more interested in the number of memory accesses, rather than number of recurses, but this number follows from our analysis. Every recurse requires k memory accesses, except when a free bucket is found. Hence we require an expected $e\,k$ accesses per flow arrival (i.e., first packet of each flow), and only k memory accesses for both (i) packets belonging to discarded flows and (ii) successive packets of sampled flows. Hence, in the worst-case scenario, where each packet belongs to a new flow (i.e., a DoS attack such as a SYN flood [22]), the per-packet cost of our measurement scheme is below $e\,k$.

Since, as explained earlier, small values of k are already practical to enhance memory usage, k can be considered to be a constant, just like e is. Therefore, our algorithm can be considered to be able to process packets linearly 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.

A series of optimizations could be considered to reduce the number of hash functions (i.e., memory accesses per packet) as the number of incoming flows severely outgrows the reservoir size. Two possible optimizations in such case would be as follows. First, we could periodically compute the largest stored flow hash. This way, packets belonging to a flow with higher hash could be immediately discarded, without checking stored hashes. Second, we could revert to $k = 1$. This would require moving each flow aggregate to the position indicated by the first hash function as new flows take its other positions. Note however that, inevitably, the algorithm would need to start with $k > 1$ to guarantee high worst-case memory usage. We argue that such optimizations, though useful, would have very limited benefit in practice, since the hardware of the monitor should still have to be dimensioned for the initial k. We further elaborate on this argument in Sec. VI.

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,

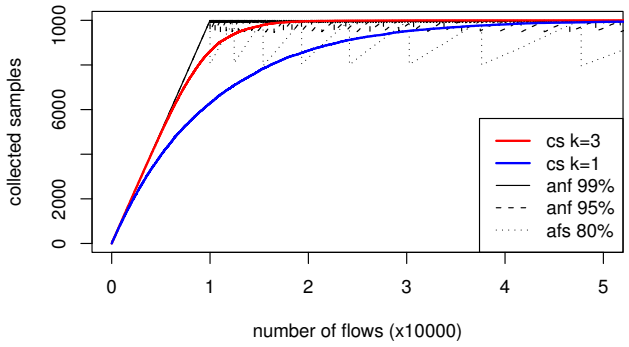

Fig. 3. Sample size obtained with several configurations.

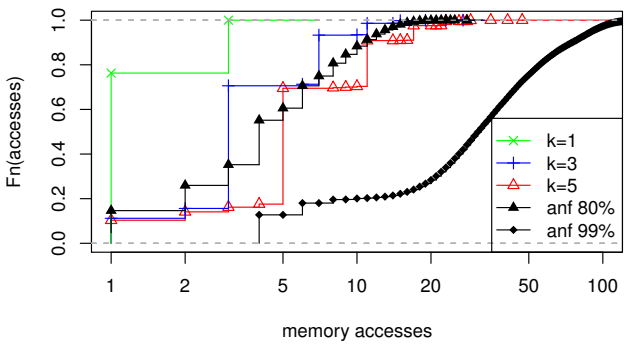

Fig. 4. CDF of the number of memory accesses with several values of k, reservoir of size 10^4 and 5 times as many flow arrivals.

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

The evaluation of our proposal is divided in two sections. In this one, we rely on synthetic traffic to understand the performance of our technique in a simple scenario. In Sec. VII, we will evaluate it with real traffic and present a few use cases that show how our technique preserves flow aggregates.

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 set an example configuration with reservoir size 10000, and generate 5 times as many 1-packet flows.

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.

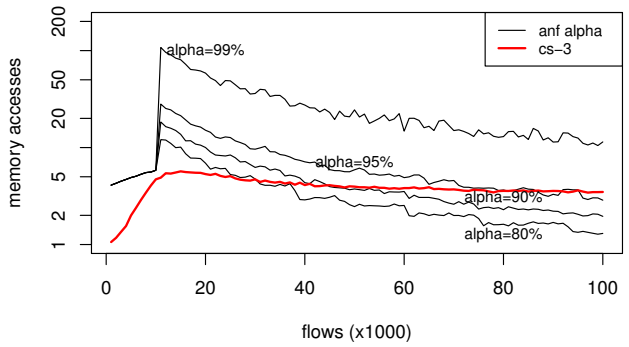

Fig. 5. Average cost per flow in bins of 1000 flows.

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 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.

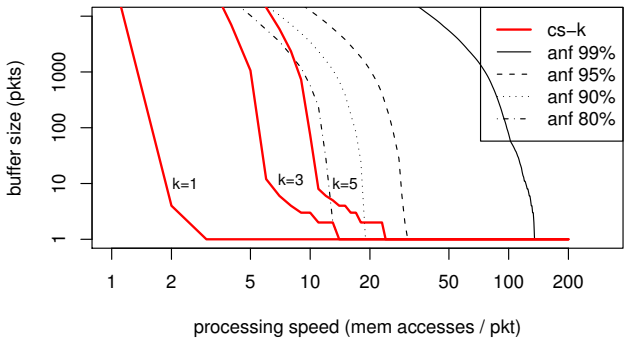

Fig. 6. Buffer size required to sample 10^4 flows as a function of the processing speed.

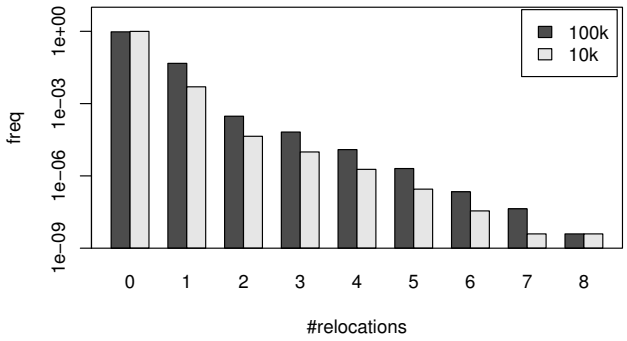

Fig. 7. Histogram of the number of relocations triggered by packets with the two reference reservoir sizes.

VII. Experiments with Real Traffic

In this section, we present experiments with real traffic. We deployed a series of CoMo [2] modules at the Gigabit access link of Universitat Politècnica de Catalunya–BarcelonaTech, which connects about 25 faculties and 40 departments (geographically distributed in 10 campuses) to the Internet through the Spanish Research and Education network (RedIRIS), and services around 50,000 users. Our experiments last 30 minutes of traffic, with around 250 million packets spread across just short of 7.9 million flows.

A. Cost

As a reference, for each flow, we computed the 5-tuple along with total number of packets and bytes (without sampling). We simultaneously ran Cuckoo Sampling with reservoirs of $r_1 = 10^5$ and $r_2 = 10^4$ flows, with $k = 3$. Average cost and its standard deviation were 2.94 and 0.40 for r_1, and 2.99 and 0.13 for r_2 (r_1 has lower cost because more packets belong to sampled flows and cause $\leq k$ memory accesses).

Note that the cost of CS is smaller than in the simulation results, where we assumed an extremely aggressive traffic mix of only one-packet flows. In this case, the average cost is slightly below the number of hash functions. This is due to the fact that a fraction of the packets find a matching flow in the first or second access. This fraction of packets is larger for r_1, since the reservoir is larger than r_2; hence its cost is slightly smaller.

Consistently with the analysis of Sec. V and the simulation results, very seldom do packet arrivals require a large number of memory accesses. Figure 7 shows the histogram of the number of relocations (note the logarithmic axis). For both r_1 and r_2, less than 13 packets triggered more than 6 relocations. Thus, a hardware pipeline could only implement 6 relocation steps with a negligible impact in the measurements.

We also ran our experiments with our implementation of Adaptive NetFlow (ANF). As has been established in Sec. VI, the cost of ANF varies with the number of incoming flows, and the monitor must be provisioned to absorb its peak cost. We computed the number of memory accesses per packet, as well as the periodic normalization. To include the cost of the normalization procedure, we uniformly spread it across the packets that arrive before the next normalization is triggered. The period of peak load then corresponds to the packets that arrive between the first and the second normalization.

Under normal traffic conditions, with reservoir size 10^5 and $\alpha = 0.95$, ANF had a worse average cost of 6.04 memory accesses per packet during this peak load period. However, as observed in Sec. VI, ANF's cost is sensitive to aggressive traffic profiles. To illustrate this issue, we ran several experiments replacing part of the traffic for a distributed denial of service attack. With a DoS using 25% of the bandwidth, this peak cost grew to 12.53; with 50%, to around 20. In both these scenarios, the average cost for Cuckoo Sampling stayed below 3 accesses per packet.

B. Collecting Traffic Aggregates

Figure 8 shows an example application of Cuckoo Sampling to estimate two kinds of flow aggregates. First, we compute the percentage of flows on each TCP port. The left figure presents the 50 ports with a highest amount of traffic, in decreasing order (*full traffic* line). We then estimate the percentage of flows using the results of gathering a sample 10^5 and 10^4 flows using both Cuckoo Sampling (CS) and Adaptive NetFlow (ANF).

Since CS is based on flow sampling, CS-based estimates are close to the real values, with the estimates obtained from the larger sample being more accurate. On the contrary, results from ANF are more error prone, given the biases introduced by packet sampling. In particular, points associated to ports where flows are larger tend to be overestimated (e.g., the first corresponds to HTTP traffic), and conversely, those that tend to be used by small ports are under-represented (e.g., the third point, which corresponds to DNS).

Figure 8 (right) presents the result of a similar experiment. This time, we group flows by internal class C subnetwork. Again, we plot the 50 subnets with largest amount of traffic, and show how these values can be correctly estimated from CS, but ANF is more error prone. Again, this is due to the biases introduced by packet sampling.

However, flow sampling methods should be considered complementary, and have to be chosen depending on the particular application or metric that the monitor is calculating. Of course, packet sampling based methods such as ANF would

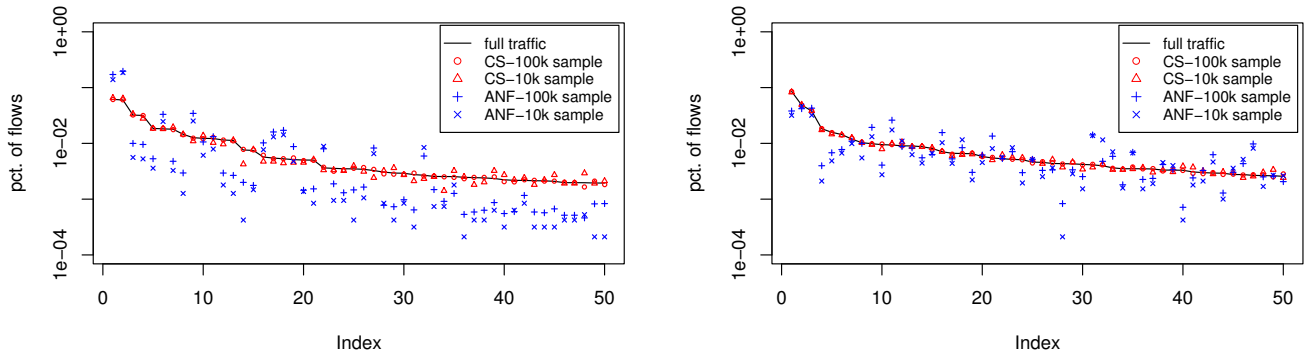

Fig. 8. Percentage of flows of the ports with largest amount of flows (left) and of the subnetworks with largest amount of flows (right).

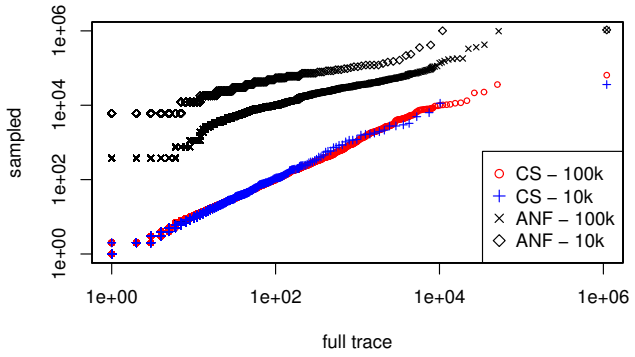

Fig. 9. QQ plots comparing actual and sampled distribution of the number of packets per flow.

estimate packet counts more precisely than flow sampling based ones.

It is well known that flow distribution estimation using packet sampling is a complex problem [23]. The main issue is that flow size distributions usually exhibit a long tail. Then, packet sampling biases measurements towards long flows, because it tends to always capture large flows and miss small ones. Figure 9 compares the actual flow size distribution against the one obtained after sampling by our method. As expected, our method correctly preserves flow size distribution, whereas ANF does not.

C. Anomaly Detection and Extraction

Another use case for our technique is anomaly detection and extraction. Recently, Ref. [24] proposed the use of a technique called frequent itemset mining to efficiently identify and present anomalous flows. Since operational networks carry large volumes of traffic, it is impossible for a human expert to review the full set of flows to discern which correspond to attention worthy events events, such as network attacks or other kinds of anomalies. An approach to ease this problem is to partition the traffic in clusters that encompass a large number of flows, and present this data for manual examination.

We use the publicly available [25] frequent item set mining tool SaM based on the Split and Merge algorithm [26], and run it using the set of TCP flows. As output, we obtain a reference data set where each cluster has at least 1% of the total flows. We then compare the results obtained with the full set of flows against those obtained from the $r_1 = 10^5$ and $r_2 = 10^5$ flow sample, collected via both CS and ANF.

The results we obtained can be observed in Table I, which shows the 10 largest (in number of flows) traffic clusters reported by the algorithm with the full sample. IP addresses have been anonymized with a prefix-preserving anonymization algorithm [27]. The table includes the ranks when running the tool from a sample of the flows. Missing entries mean that the frequent itemset mining tool has failed to report a cluster.

The algorithm was able to identify all but one of the clusters from flow-sampled traffic. Note that the samples are quite small; the effective sampling rate for r_1 was around 1.27%, and for r_2, around 0.12%. Thus, the time required to extract the clusters was under 0.1 seconds for the smallest sample and around 1 second for the larger one, compared to around 85 seconds to process the full set of flows.

When running the algorithm on the flow data reported by Adaptive NetFlow, the algorithm is unable to identify the vast majority traffic clusters. This is a well expected result that is due to the fact that ANF relies on packet sampling, which tends to miss many small flows, and is skewed towards large flows.

VIII. 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 23 lines of pseudo-code

TABLE I
TRAFFIC CLUSTERS THAT ENCOMPASS THE LARGEST AMOUNT OF FLOWS, WITH ANONYMIZED IP ADDRESSES

src	sport	dst	dport	proto	#flows	rank	rank in sample: flow samp. 10^5	flow samp. 10^4	pkt samp. 10^5	pkt samp. 10^4
79.205.92.135	*	171.52.0.0/16	22	TCP	3.8%	1	1	1	7	–
5.135.48.203	*	171.52.0.0/16	1433	TCP	3.1%	2	3	4	29	–
171.52.0.0/16	1433	5.135.48.203	*	TCP	3.1%	3	2	2	51	–
171.52.175.235	*	116.23.115.107	80	TCP	2.8%	4	5	3	1	4
116.23.115.107	80	171.52.175.235	*	TCP	2.8%	5	4	5	2	5
138.0.0.0/8	*	171.52.0.0/16	*	TCP	2.5%	6	6	6	6	–
66.233.148.83	*	171.52.0.0/16	32000	TCP	2.5%	7	8	7	–	–
171.52.0.0/16	*	138.0.0.0/8	*	TCP	2.4%	8	9	10	4	–
*	*	171.52.0.0/16	80	TCP	2.4%	9	7	–	3	–
171.52.87.0/24	*	*	*	TCP	2.3%	10	10	12	–	–

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 of 1-packet flows that mimicked a distributed denial of service attack, as well as regular traffic. As a conclusion of this work, we believe this method to be very practical and, in particular, to be suitable for implementation within routers.

AVAILABILITY

An implementation of Cuckoo Sampling can be downloaded from http://people.ac.upc.edu/jsanjuas/cuckoo_sampling/index.html.

ACKNOWLEDGMENTS

This research was funded by the Spanish Ministry of Science and Innovation under contract TEC2011-27474 (NOMADS project), and by the *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, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta, "Load shedding in network monitoring applications," in *Proceedings of USENIX Annual Technical Conference*. Usenix Association, Jun 2007, pp. 59–72.
[3] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," *SIGCOMM Comput. Commun. Rev.*, vol. 34, pp. 245–256, August 2004.
[4] R. Kompella and C. Estan, "The power of slicing in internet flow measurement," in *Proceedings of the 5th ACM SIGCOMM 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, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, p. 245, Oct. 2004.
[7] N. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Transactions on Networking*, vol. 9, no. 3, pp. 280–292, Jun. 2001.
[8] N. Hohn and D. Veitch, "Inverting sampled traffic," in *Proceedings of the 3rd ACM SIGCOMM IMC*, no. 1. ACM, Feb. 2003.
[9] N. Duffield, C. Lund, and M. Thorup, "Charging from sampled network usage," *Proceedings of the First ACM SIGCOMM Workshop on Internet Measurement Workshop - IMW '01*, p. 245, 2001.
[10] T. Zseby, T. Hirsch, and B. Claise, "Packet sampling for flow accounting: Challenges and limitations," 2008.
[11] J. Mai, A. Sridharan, C. Chuah, H. Zang, and T. Ye, "Impact of packet sampling on portscan detection," *Selected Areas in Communications, IEEE Journal on*, vol. 24, no. 12, pp. 2285–2298, Dec. 2006.
[12] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang, "Is sampled data sufficient for anomaly detection?" *Proceedings of the 6th ACM SIGCOMM IMC*, p. 165, 2006.
[13] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina, "Impact of packet sampling on anomaly detection metrics," in *Proceedings of the 6th ACM SIGCOMM IMC*, 2006.
[14] N. Duffield, C. Lund, and M. Thorup, "Properties and prediction of flow statistics from sampled packet streams," *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment - IMW '02*, p. 159, 2002.
[15] ——, "Estimating flow distributions from sampled flow statistics," *Proceedings of ACM SIGCOMM '03*, 2003.
[16] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting: Focusing on the Elephants , Ignoring the Mice," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 270–313, 2003.
[17] C. Barakat, G. Iannaccone, and C. Diot, "Ranking flows from sampled traffic," in *Proceedings of ACM CoNext*, 2005.
[18] E. Cohen, N. Grossaug, and H. Kaplan, "Processing top-k queries from samples," *Computer Networks*, vol. 52, no. 14, pp. 2605–2622, Oct. 2008.
[19] N. Duffield, C. Lund, and M. Thorup, "Flow sampling under hard resource constraints," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, p. 85, Jun. 2004.
[20] J. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985.
[21] R. Pagh, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.
[22] J. Lemon *et al.*, "Resisting syn flood dos attacks with a syn cache," in *Proceedings of the BSDCon*, 2002, pp. 89–97.
[23] P. Loiseau, P. Gonçalves, S. Girard, F. Forbes, and P. Vicat-Blanc Primet, "Maximum likelihood estimation of the flow size distribution tail index from sampled packet data," in *Proceedings of the 11th int. conf. on Measurement and modeling of computer systems*, ser. SIGMETRICS '09, 2009.
[24] D. Brauckhoff, X. Dimitropoulos, A. Wagner, and K. Salamatian, "Anomaly extraction in backbone networks using association rules," in *Proceedings of the 9th ACM SIGCOMM IMC*. ACM, 2009, pp. 28–34.
[25] C. Borgelt, "SaM frequent item set mining tool," http://www.cisco.com/web/go/netflow.
[26] C. Borgelt and X. Wang, "SaM: A split and merge algorithm for fuzzy frequent item set mining," 2009.
[27] J. Xu, J. Fan, M. Ammar, and S. Moon, "Prefix-preserving ip address anonymization: Measurement-based security evaluation and a new cryptography-based scheme," in *Proceedings of 11th IEEE Conf. on Network Protocols*, 2002.