CELL: Counter Estimation for Per-flow Traffic in Streams and Sliding Windows

Rana Shahout
Technion

Roy Friedman
Technion

Dolev Adas
Technion

***Abstract*—Measurement capabilities are fundamental for a variety of network applications. Typically, recent data items are more relevant than old ones, a notion we can capture through a sliding window abstraction. These capabilities require a large number of counters in order to monitor the traffic of all network flows. However, SRAM memories are too small to contain these counters. Previous works suggested replacing counters with small estimators, trading accuracy for reduced space. But these estimators only focus on the counters' size, whereas often flow ids consume more space than their respective counters. In this work, we present the CELL algorithm that combines estimators with efficient flow representation for superior memory reduction.**

We also extend CELL to the sliding window model, which prioritizes the recent data, by presenting two variants named RAND-CELL and SHIFT-CELL. We formally analyze the error and memory consumption of our algorithms and compare their performance against competing approaches using real-world Internet traces. These measurements exhibit the benefits of our work and show that CELL consumes at least 30% less space than the best-known alternative. The code is available in open source.

I. Introduction

Network statistics is a vital tool in network engineering [31], load balanced routing [37], [52], network security [16] and anomaly detection [18]. Such applications require algorithms that are both time and space efficient to cope with the ever-increasing line rates. Previous works suggested replacing counters with small estimators, trading accuracy for reduced space. Such space reduction enables fitting more counters into SRAM memory [54] (or TCAM [41]), which tends to be much smaller than the abundant DRAM memory.

A stream of packets flowing through the network in general or a specific switch can be divided into multiple *flows*, each with its unique identifier. For example, the 5-tuple (source_ip, source_port, destination_ip, dest_port, protocol) is often used as a flow identifier, but sometimes it can be just the (source_ip,source_port) etc. A fundamental network statistics problem is the *flow's frequency*, i.e., how many packets with the given flow's identifier passed through the network/switch.

The challenge is that in a modern network the number of flows can be huge, and precisely representing each flow's frequency requires maintaining at least one counter per flow as well as the flow's identifier. Hence, a naive approach would consume much more memory than the typical available SRAM memory. This has been largely addressed by efficient counter representations [23], [33], [34] and sketches [10], [11], which reduce memory consumption by lowering the fidelity of the statistics, i.e., making it approximate. However, sketches only guarantee an error that is proportional to the entire stream size.

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

Another line of solutions is protocols such as MG [38], Frequent [28], SpaceSaving [36], and RAP [5], which maintain a fixed-size table of flow ids and associated counters with some rules on which flows are represented in the table. But, their error guarantee is also proportional to the entire stream size.

In traditional hardware implementations, SRAM memory tends to be very tight and constant. Hence, for a given memory budget, the goal is to obtain the minimal approximation error. On the other hand, there is a recent trend to increasingly implement network functions in software. In particular, when hosting multiple VMs on a single physical host, the host needs to provide a virtual switch capability between the VMs and the network (and among themselves). Yet, software implementations can better handle dynamic memory management than their hardware counterparts, and SRAM memory is not as small as it used to be. Hence, it becomes interesting to exploit these abilities, and design solutions that minimize the amount of memory required for a given approximation error budget.

Recent data items are often more relevant than old ones, so many applications realize this through a *sliding window* abstractions. In the sliding window model [12] only the most recent items in the stream are considered, while older ones do not affect the quantity we wish to estimate. Indeed, the problem of maintaining various types of sliding window statistics was extensively studied [3], [6], [12], [30], [42].

In this work, we focus on counter estimation techniques [39]. Here, the goal is to associate each flow with a frequency counter, which counts how many packets from that flow have arrived as part of a given stream. To decrease the space consumption of counters, their accuracy is being reduced. For example, suppose we increment a counter c with probability $1/r$ on each packet arrival. Whenever queried for the value of c, we would return $c \times r$. In this case, the expected returned value of c is the same as the real value of c, except that now each specific returned value might contain an error. The benefit is that in order to represent a value v, we only require $\log \frac{v}{r}$ bits instead of $\log v$ bits required to accurately represent v. Further, the increment probability of a counter whose value is c might depend on c, in which case the evaluation function becomes more involved.

To maintain statistics about k flows, a naive approach would allocate a counter per flow. At the extreme, for a stream of

N packets, we might need up to N counters. Yet, if the counters of many of these flows hold the same value, an additional space reduction can be obtained by enabling all flows whose frequency estimation is the same to share the same counter [50]. Still, one needs to maintain the identifiers of all flows in order to map them to their respective shared counter. However, flows identifiers are often much larger than their counters. Hence, additional savings are called for.

Our work combines ideas from sketches with counter estimation. We also extend our ideas to the sliding window model.

Our Contributions: Our first contribution is an algorithm called Counter Estimation Levels List (CELL), a novel counter estimation technique that reduces space consumption by compacting flows identifiers and using estimators. As illustrated in Figure 1, CELL has two representations, Approximate Membership (AM) and Hash Table (HT). CELL maps the flows to levels according to their frequencies. Each flow starts at the first level and probabilistically climbs to higher levels as more packets arrive. It stores flow fingerprints either in levels of approximate memberships (AM) or in a hash table along with their level number. Next, it uses the level number to compute frequencies using estimators.

Second, we propose two efficient algorithms, *Randomized Counter Estimation Levels List* (RAND-CELL) and *Shift Counter Estimation Levels List* (SHIFT-CELL), both extending CELL to the sliding window model by probabilistically maintaining the most recent W elements.

With sliding windows, we need to remove the oldest element upon a new arrival. RAND-CELL randomly seeks a prior element arrival and decrement one from the corresponding flow's estimation. In contrast, SHIFT-CELL applies batch reductions on all flows every C arrivals as shown in Figure2. Finally, we compare our algorithms to state-of-the-art approaches. Specifically, we compare CELL to CEDAR [51] and ICE-Buckets [14]. While CEDAR and ICE-Buckets are more accurate, we show that CELL is considerably more space efficient. For the sliding window model, we compare RAND-CELL and SHIFT-CELL to SWAMP [4] and show that we reduce the memory consumption by at least 4x with windows of above $100,000$ items (on the order of 0.1 seconds on modern networks). SWAMP's significant memory consumption largely comes from the fact that it must store the fingerprints of all window elements, in contrast with our work whose memory overhead depends mostly on the number of unique flows.

We note that WCSS [6] also estimates item frequencies (in constant time) over a sliding window. It guarantees an additive error of $W\epsilon$ for a window size W and a configuration parameter ϵ. In contrast, the error of RAND-CELL is a *relative error* and is smaller for small flows as proved in Section V-A.

II. Related Work

1) Full Size Counters: Hybrid counters, proposed in [45], [47], store in SRAM only the least significant bits of each counter and in DRAM the remaining bits of the counters. This enables fast counter updates as they occur mostly in

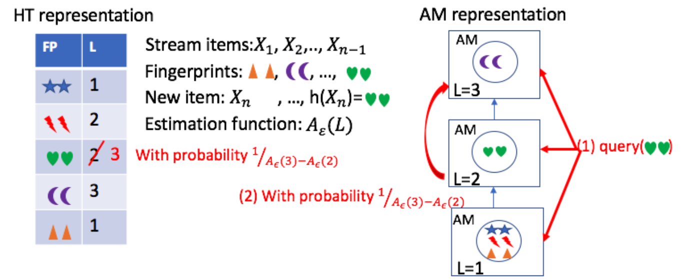


Fig. 1: An overview of CELL with the two representations (AM and HT): Given estimation function $A_\epsilon(L)$, upon item x_n arrival: in HT, the level of the element's fingerprint is incremented with probability $\frac{1}{A_\epsilon(3)-A_\epsilon(2)}$. In AM, the first step is to locate the element level, then its fingerprint moves to the next level with probability $\frac{1}{A_\epsilon(3)-A_\epsilon(2)}$.

SRAM, but slow reading as it involves DRAM access. CounterBraids [33] employs counters compression to obtain a small representation whose encoding can occur entirely in SRAM. However, decoding is cumbersome and slow, resulting in slow counter reads. Likewise, a related algorithm, Brick [24], is an online counter array that encodes variable-length counters by restricting the counter sum. But it still has limited counting capacity as the average counter value increases. In this paper, we are interested in fast operations.

2) Counter Approximations: The counter approximations approach suggests replacing counters with smaller estimators. Such reduction enables more counters to fit into SRAM at the cost of reduced accuracy. Approximate Counting [40] was the first such work. Later, Small Active Counters (SAC) [49] adapted this idea to networking. DISCO [23] provided variable-sized increments and improved accuracy. Further, [22] introduced a way to increase the relative error as the counters grow, so they can offer unlimited capacity.

CEDAR [51] provided a recursive estimation function for estimators' increase and proved that it is optimal. Besides, CEDAR proposed an *up-scaling* algorithm for adjusting the relative error of the entire counter scale through an upscaling algorithm. Up-scaling is performed according to the maximal counter without any assumption on flow distribution.

ICE-Buckets [14] reduces the amortized error by efficiently utilizing multiple counter scales. It partitions the flows into buckets and configures the optimal estimation function according to each bucket's counter scale. I.e., with ICE-Buckets, the error in each bucket is proportional to the largest counter in that bucket rather than the largest counter overall.

3) Counter Estimation over Sliding Window: Estimating item frequencies over sliding windows was first studied in [3]. They proposed an algorithm that requires $O(\epsilon^{-1}\log^2\epsilon^{-1}\log W)$ bits within a $W\epsilon$ additive error over a W sized window. Then [30] reduced it to the optimal $O(\epsilon^{-1}\log W)$ bits. [26] improved the update time to $O(1)$ while being able to find all heavy hitters in the optimal $O(\epsilon^{-1})$ time. Further, [3], [30] considered window expanding and shrinking when processing updates.

Sliding Sketch [20] is a generic framework that can adapt any sketch that follows the k-hash model to the sliding window

model. The main idea behind Sliding Sketch is to divide the sketch into many "time zones" and to delete out-of-date information through scanning. However, these sketches only guarantee an error proportional to the entire stream size while estimators guarantee a relative error.

The Sliding Window Approximate Measurement Protocol (SWAMP) [4] is a sliding window algorithm that solves per-flow counting. SWAMP uses a single hash function (h) that maps flow-id to a fingerprint. The SWAMP data structure contains a cyclic fingerprint buffer of length W (the window size), denoted CFB, which stores fingerprints. Fingerprints are also stored in TinyTable [15], which provides efficient multiplicity information. SWAMP requires linear memory and counts accurately with a probability of at least $1-\delta$.

4) Sketch Based Techniques: Sketches like Count Sketch [10], Space Saving (SS) [36] and Count Min Sketch [11] are popular for maintaining item's frequency estimation over a stream. The sketching and sampling-based algorithms produce an additive error on the order of $N\epsilon$, where ϵ is a predetermined constant and N is the total count size. Elastic Sketch [53] divides packet processing into two parts: heavy and light. The light part is a Count Min Sketch that tracks mice flows, and the heavy part is a hash bucket table for the heavy flows that are stored and evicted to the light part as needed. Elastic Sketch promises an additive error on the order of $N_l\epsilon$ where N_l is the size of the sub-stream recorded by the light part while estimators promise a relative error. Ada-Sketches [48] have better error guarantees than straightforward sketches design, but it is only for recent items, while old items even have more errors than new ones.

5) Approximate Memberships: The most famous approximate membership structure is Bloom filter [7]. Several Bloom filter variants were suggested to improve various aspects of Bloom filters, such as Blocked Bloom filter (BlockedBF) [43], [44], Balanced Bloom filters (BalancedBF) [27], Counting Bloom filters (CBF) [29] and Inverted Bloom filter [19]. TinySet [13] is an alternative construction based on a fingerprint hash-table. It is therefore more space-efficient than Bloom filters for similar false-positive rates.

A fingerprint hash-table was first suggested in d-left hashing [8], [9]. Similarly, Cuckoo hashing [35] calculates a perfect hash function using the power of two choices. TinyTable [15] is another recent compact hash table based construction.

III. Preliminaries

Given a *universe* $\mathcal{U}$, a stream $\mathcal{S} = x_1, x_2, \ldots, x_N \in \mathcal{U}^*$ is a sequence of universe elements. Denote N the number of items (network packets) in $\mathcal{S}$ and MF the maximal number of unique flows ($MF \leq N$). Each element is associated with a flow label f. We define the size of flow f in $\mathcal{S}$ as the number of elements with flow label f in the stream $\mathcal{S}$. We denote this quantity by N_f and the maximum flow size by MFS. See Table I for summary of these notations.

(ϵ, δ)-Per-flow counting: We consider algorithms that answer queries for flow size estimation in the stream $\mathcal{S}$ while guaranteeing a given error ϵ with a probability of at least $1-\delta$. Such an algorithm must support the following operations:

- **ADD(x)**: given an element $x \in \mathcal{U}$, append x to $\mathcal{S}$.
- **QUERY(f)**: return an estimation $\widehat{N_f}$ for N_f such that $\left|\widehat{N_f} - N_f\right| \leq \epsilon N_f$ with probability $1-\delta$.

In the sliding window model, we denote by $W \in \mathbb{N}$ the *maximal window size*. In this case, $MF \leq W$. At any time point t, i.e., when the element x_t arrives, the sliding window maintains the most recent W elements in $\mathcal{S}$: $x_{\max\{0,t-W+1\}}, \cdots, x_t$, denoted $Win(t, W)$.

Given a flow f and a maximal window size W, the *flow size of f in W*, denoted N_f^W, is the number of elements with flow label f that appear within the last W elements of $\mathcal{S}$.

(W, ϵ, δ)-Per-flow window counting: We consider algorithms that answer queries for flow size estimation in the last W elements window while guaranteeing a given error ϵ with a probability of at least $1-\delta$. In particular, such algorithms must support the following operations:

- **ADD(x)**: given an element $x \in \mathcal{U}$, append x to $\mathcal{S}$.
- **QUERY(f)**: return an estimation $\widehat{N_f^W}$ for N_f^W such that $\left|\widehat{N_f^W} - N_f^W\right| \leq \epsilon N_f^W$ with probability $1-\delta$.

Given an error parameter ϵ_M, an *approximate membership* data structure is a randomized data structure that represents a set $\mathcal{A}$ and answers whether an element x is in the set $\mathcal{A}$ ($\mathcal{A} \subset \mathcal{U}$) with the following guarantees:

- If $x \in \mathcal{A}$, the answer is always true (no false negative).
- If $x \notin \mathcal{A}$, the answer is false with probability at least $1-\epsilon_M$ and true (false positive) with probability at most ϵ_M.

We consider the following key performance metrics:

Estimation Accuracy: We discuss the quality of the estimation in terms of *root mean squared relative error (RMSRE)*: Denote $\hat{n}$ the random variable of the estimation flow size after n elements have arrived

$$RMSRE[n] = \sqrt{E[(\frac{\hat{n}-n}{n})^2]}.$$

The *maximum relative error* is $\max_{n \leq N} RMSRE[n]$.

Storage Overhead: Memory consumption that guarantees a given error ϵ.

Intuitively, RMSRE captures the bound on the expected relative error, i.e., the difference between the actual and estimated frequencies of any flow. Our goal is to minimize memory requirements in order to achieve the above accuracy goals. As mention in the introduction, in this work we view the RMSRE as a hard constraint, whereas the memory usage is the optimization parameter.

IV. *Counter Estimation Levels List* (CELL)

A. Algorithm

In this section, we describe our base algorithm, CELL, which solves the per-flow counting problem. CELL architecture is based on multiple levels of estimators and a memory-efficient mapping from flow to a level. The estimation value for level i is calculated with an optimal estimation function A

TABLE I: List of Symbols

Symbol	Meaning
$\mathcal{S}$	the data stream
$\mathcal{U}$	the universe of elements (all possible type of packets)
N	number of elements in the stream
n	number of elements that have arrived
MF	the maximal number of unique flows
MFS	the maximum flow size
N_f	the size of flow f in $\mathcal{S}$
$\widehat{N_f}$	an estimation of N_f
W	the maximal window size
N_f^W	the size of flow f in the last W elements of $\mathcal{S}$
$\widehat{N_f^W}$	an estimation of N_f^W
ϵ	estimation accuracy parameter of per-flow counting algorithm
ϵ_M	estimation accuracy parameter of approximate membership
δ	probability of failure to provide the accuracy guarantee
$A_\epsilon(l)$	estimation function
E_i	estimation value of level i (=$A_\epsilon(i)$)
L	levels number
$node_i$	the data structure associated with level i in the AM representation

that accepts a level as input and returns an estimation value. We use the estimation function proved to be optimal in [14]:

$$A_\epsilon(l) = \frac{(1+2\epsilon^2)^l - 1}{2\epsilon^2}(1+\epsilon^2). \quad (1)$$

For example, if a flow f is mapped to Level i, then the frequency of f is estimated to be $A_\epsilon(i)$; denote this value by E_i. When a new flow f arrives, it is assigned to the first level with probability $\frac{1}{E_1}$. On each additional arrival, flow f at level i is advanced to level $i+1$ with a probability $\frac{1}{E_{i+1}-E_i}$. Abstractly, the first flow that reaches a given level can be viewed as implicitly initiating it in a manner that depends on additional design aspects that are elaborated below.

As shown in Figure 1, CELL has two representations according to the mapping from flows to estimators' levels. Below we introduce these two representations.

***Hash Table Representation**:* Store a mapping from a flow's fingerprint to its level using a memory-efficient hash table like Cuckoo hash table [35] or TinyTable [15]. In Figure 1, the flow of item X_{n-W+1} is mapped to level 3.

***Approximate Membership Representation**:* For each level, we represent this level's flows by an approximate membership structure that supports deletions, such as TinySet [13] or other variants of counting bloom filters [17], [25], [29], [32], [46] (i.e., variants of Bloom filters that support deletions). Here, to lookup a flow, we can scan all these structures from the top-level downward until obtaining a positive indication. As illustrated in Figure 1, the approximate membership structure of level 3 contains the flow of item X_{n-W+1}. To make this efficient, we configure the approximate membership structures of all levels to use the same hash functions. This way, the hash functions, whose computation is the heaviest, are computed only once on each lookup regardless of the number of levels. A benefit of using TinySet is that even if the TinySet instance of each level is sized according to the expected number of flows at that level (bounded by $\frac{N}{E_{level}}$), we can still utilize the same single hash function for all [13].

Since we have no hard limit on the number of levels, some levels might not be associated with any flows, a doubly-linked list is an obvious candidate. Whenever a flow is transferred to a level i whose node is not allocated, it is first allocated and then inserted into its corresponding place in the list. Similarly, when a given level no longer holds any flows, its node can be removed from the list and freed.

Yet, when most levels have flows mapped to them and therefore allocated, the memory overhead of the linked list pointers can become significant. Also, scanning a dynamically allocated linked list is not CPU cache-friendly.

Another alternative is to hold the levels in a continuous array. But, as the number of levels varies, we may occasionally need to reallocate the array, a potentially expensive operation. A naive approach is to pre-allocate a very large array, but it is memory wasteful. An intermediate solution is to maintain the levels in a linked list of arrays, for a flexible tradeoff between the benefits and drawbacks of both approaches.

In both approaches, we employ approximate data structures. Thus, we may obtain a false positive indication that a given flow f is associated with a given level i even though f has never arrived, or f is at a lower level. Denote this false positive probability by ϵ_M. In equation (5) we calculate how large ϵ_M should be so that with probability $1-\delta$, we end in the right level of flow f. In our paper, we experiment with TinyTable and Cuckoo hash as representatives of the first approach and with TinySet as a representative of the second approach.

B. CELL Analysis

1) Memory vs. Error in CELL: The memory consumption of CELL is influenced by two factors: the number of estimator levels and the representation cost of the mapping from flows to levels. When using Equation 1 for the estimators' values, there is no need to allocate any memory for them as we use the level number as an input to A_ϵ. Yet, when queries are frequent, it may be computationally preferable to pre-compute the estimator values for all levels, in which case each value consumes the size of a float (or a double).

Hash Table Representation: For hash table (HT) representation, without any knowledge of the stream's distribution, the worst case is that every flow arrives only once. Hence, the HT needs to be sized so it can store up to N items. In the case of Cuckoo hash [35] and TinyTable [15], this means having roughly $1.1-1.2N$ entries. The number of bits required to encode the level of a flow is logarithmic in the maximal number of levels L. Due to TinyTable's self-adjusting counters, if the table was sized to store N items, it can already accommodate the counters within the same space. In contrast, Cuckoo would require allocating a $\log L$ bits counter for each entry. As L itself is bounded by $\log N$, we get that each counter in a Cuckoo HT would require $O(\log\log N)$ bits.

As for the impact of the false-positive ratio ϵ_M, the number of flow's fingerprint bits is logarithmic in $\frac{1}{\epsilon_M}$. This gives a total memory overhead of $N(\log\frac{1}{\epsilon_M} + \log\log N)$ in the case of a Cuckoo HT and $N(\log\frac{1}{\epsilon_M})$ in the case of TinyTable[1].

Approximate Membership Representation: In approximate membership (AM) representation, each level requires an AM structure. Again, without any knowledge regarding the

[1]Recall that when storing full flow identifiers, each flow-id can easily reach 96 bits for IPv4 and almost double that with IPv6.

stream's distribution, an AM for level l must be sized to store up to $\frac{N}{E_l}$ elements. Using TinySet, this means a little bit more than $\frac{N}{E_l}$ entries [13] in each AM structure. Similarly to before, for a false positive ratio ϵ_M, we need the fingerprints to be $\log\frac{1}{\epsilon_M}$ long. Hence, the total memory required by each level is $O(\frac{N}{E_l}\log\frac{1}{\epsilon_M})$. Since L is bounded by $\log N$, we get a total memory complexity of $O(\frac{N}{E_l}\log N\log\frac{1}{\epsilon_M})$ bits.

Obviously, the AM representation is less memory efficient that HT, it is important for the extensions of CELL to sliding windows discussed in Sections V and VI below. In those extensions, the AM based realizations of the algorithms are more computationally efficient and intuitive.

2) The Number of Levels Generated by CELL: Denote L the maximal level. Recall that MFS is the maximum flow size. Denote by $EMFS$ the estimation of MFS, by $\mathbb{E}EMFS$ the expected $EMFS$. Let $MEFS$ be the maximal estimated flow size and $\mathbb{E}MEFS$ be the expectation of $MEFS$. Using equation (1):

$$\mathbb{E}MEFS = \frac{(1+2\epsilon^2)^L - 1}{2\epsilon^2}(1+\epsilon^2) \tag{2}$$

Extracting L:

$$L = \log_{1+2\epsilon^2}\left(\frac{2\epsilon^2 \cdot \mathbb{E}MEFS}{1+\epsilon^2} + 1\right) \tag{3}$$

As proved in CEDAR [51], $\mathbb{E}EMFS = MFS$. We need to prove that $\mathbb{E}MEFS \approx \mathbb{E}EMFS$ to get that $EMEFS \approx MFS$. We denote by $F(l)$ the random variable that represents the estimation of MFS that reside in level l. We can divide $F(l)$ i.i.d geometric random variables $\widehat{F}(p_i)$ with parameter $p_i = \frac{1}{E_i - E_{i-1}}$, $i = [0, \cdots, L-1]$, $\mathbb{E}EMFS = \mathbb{E}(F(l)) = \mathbb{E}(\sum_{i=0}^{L-1}\widehat{F}(p_i)) = \sum_{i=0}^{L-1}\mathbb{E}(\widehat{F}(p_i)) = \sum_{i=0}^{L-1} p_i = E_L = MEFS$. So with high probability, we have:

$$L = \log_{1+2\epsilon^2}\left(\frac{2\epsilon^2 \cdot MFS}{1+\epsilon^2} + 1\right) \tag{4}$$

3) CELL's Probabilistic Guarantee:

a) Hash Table (HT) Representation: When using a fingerprint hashtable, we can satisfy the error bound ϵ with probability $1-\epsilon_M$ by configuring the fingerprint size such that its false positive probability is bounded by $1-\epsilon_M$. This is obtained when the size of the fingerprint is $\lceil\log\epsilon_M^{-1}\rceil$.

b) Approximate Membership (AM) Representation: Let f_l be a flow that resides at level l out of L levels in total. Recall that according to CELL, we search f_l from the top level, L, downward. If the AM of level $i > l$ returns true, according to our algorithm we return $A_\epsilon(i)$. Therefore, $A_\epsilon(l)$ is evaluated as the estimation of f_l's frequency only if all the AMs of levels above l return false. This means that we get an error of ϵ with a probability of $(1-\epsilon_M)^{L-l}$. In the worst case, the probability is $(1-\epsilon_M)^L$ for the lowest level.

In other words, to obtain a probabilistic error bound of δ, we need to size the fingerprints of TinySet so that the AM of each level is ϵ_M such that $1-\delta \geq (1-\epsilon_M)^L$. We have that $(1-\delta)^{\frac{1}{L}} \geq 1-\epsilon_M$. Hence, we should set

$$\epsilon_M \geq 1-(1-\delta)^{\frac{1}{L}} \tag{5}$$

where L is according to equation (4).

Algorithm 1 CELL Algorithm (AM representation)

```
Initialization:
   initialize  head = linked-list of approximate membership with estimate values,
   initialize  tail = pointer to the last node of the linked-list.
Initialization: nLevels ← 1
 1: function ADDLEVEL
 2:    Initialize new node and link it to the tail node
 3:    nLevels ← nLevels + 1
 4: function ADD(x_i)
 5:    f_x ← ComputeFlow(x_i)
 6:    node_i ← tail
 7:    while node_i.A.TEST(f_x) = false do
 8:       node_i ← previous node
 9:    if node_i == node_0 then
10:       node_0.A.ADD(f_x)
11:    else
12:       with probability P_i = 1/(E_{i+1} − E_i):
13:          if i + 1 < nLevels then
14:             AddLevel()
15:          node_i.A.DELETE(f_x)
16:          node_{i+1}.A.ADD(f_x)
17: function QUERY(f)
18:    node_i ← tail
19:    while node_i.A.TEST(f) = false do
20:       if node_i == node_0 then return 1
21:       node_i ← previous node
       return node_i.E_i
```

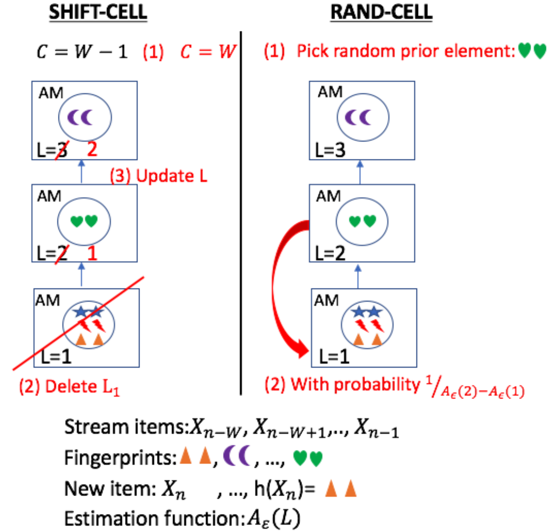


Fig. 2: Illustrating RAND-CELL and SHIFT-CELL in AM presentation. Upon x_n arrival: RAND-CELL randomly seeks a prior element arrival and decrement one from the corresponding flow's estimation. SHIFT-CELL applies batch reductions on all flows every C arrivals.

V. The RAND-CELL Algorithm

Randomized Counter Estimation Levels List (RAND-CELL) adapts the base algorithm CELL to the sliding window model in a randomized manner. That is, RAND-CELL addresses the per-flow window counting problem, by probabilistically maintaining the most recent W elements and measuring per-flow traffic for them using a CELL structure. The basic concept is generic, but for clarity, we explain it for the AM based implementation of CELL.

Ideally, when a new element arrives at time point t, we aspire to remove the oldest element (at time point $t-1$) from CELL's data structures and then update CELL with the newly arriving element. Alas, precisely removing the oldest element

Algorithm 2 RAND-CELL Algorithm

```
    Initialization:
      initialize  Base = instance of the base algorithm CELL,
    Initialization: arrivals ← 0, cycle ← Base.firstLevel()
 1: function ADD(x_i)
 2:     select an index uniformly at random
 3:     j ← 0
 4:     while Σ_{k≤j}(n_k E_k) ≤ index do
 5:         j ← j + 1
 6:     select flow uniformly from level j
 7:     with probability P_j = 1/(E_j − E_{j−1}):
 8:         Base.node_j.A.DELETE(f_x)
 9:         Base.node_{j−1}.A.ADD(f_x)
10:     Base.Add(x_i)
11: function QUERY(f) return Base.Query(f)
```

requires remembering the exact order of all elements in the window W. This requires an additional $O(W)$ space, which becomes costly when W is large as in SWAMP [4]. This is why we resort to the following probabilistic approach.

Upon x_t arrival, we add it to the first level with probability of $\frac{1}{E_1}$. Next, we seek a prior element arrival at random and deduct 1 from its estimation. Denote by j the level of flow f, when using CELL, decrementing 1 from the estimation of flow f is analogous to moving f to level $j-1$ with probability $\frac{1}{E_j-E_{j-1}}$ (the opposite of incrementing the estimation by 1) where $E_j = A_\epsilon(j)$ (see Table I).

The challenge in fully realizing this idea stems from the fact that we only maintain flows' estimations rather than individual arrivals. Hence, to mimic choosing a random element arrival for removal, we select uniformly at random an index i, $i \in [0 \dots (W-1)]$. We then skip to the jth level such that $\Sigma_{k\leq j}(n_k E_k) \leq i$ but $\Sigma_{k\leq j+1}(n_k E_k) > i$, where n_k is the number of flows in the kth level. Then we select a flow uniformly at random in j and move it to level $j-1$ with probability $\frac{1}{E_j-E_{j-1}}$ as illustrated in Figure2.

In the worst case, the above requires scanning all the levels on every element arrival. But, as mentioned, the levels' number can be bounded by $O(logW)$, so scanning all levels is not a heavy operation. We can also maintain partial summaries for a further logarithmic access time reduction.

Intuitively, the above approximates a sliding window because in each random removal step, a prior arrival is chosen with a probability that is roughly $1/W$. Hence, in expectation, such a prior arrival will be removed after W random selections. We formalize this notion in the correctness proof below.

Observation 1. *In the sliding window model, given window size W, $L = O(\log W)$, where L is the number of levels*

In the sliding window model, according to (4), even when $MFS = W$ we get $L = O(\log W)$.

A. RAND-CELL Correctness

It is easy to verify that in expectation, after every set of W elements, the total estimates for all flows held by CELL (or RAND-CELL) remains W: Given time point t, denote $\widehat{N_t^W}$ the total estimates till time point t within the last W. Consider a sliding window $Win(t, W)$: $\mathbb{E}(\sum \widehat{N_t^W}) = W$.

At time $t+1$, a new element x_{t+1} just arrived. RAND-CELLadds it to the first level with probability $\frac{1}{E_1}$ and then chooses a flow from some level j and moves it to level $j-1$ with probability $\frac{1}{E_j-E_{j-1}}$ (if $j = 1$ we delete the flow from level j with probability $\frac{1}{E_j}$). So the expected value of the total estimates between $Win(t+1, W)$ and $Win(t, W)$ is:

$$\mathbb{E}(\sum \widehat{N_{t+1}^W} - \sum \widehat{N_t^W}) = \frac{1}{E_1} E_1 - \frac{1}{E_j - E_{j-1}} (E_j - E_{j-1}). \quad (6)$$

That is, the total estimates remain W after $W+1$ arrivals.

Theorem 2. *An arrival x_t is eliminated in expectation after W arrivals*

Proof. Denote by i the number of elements that arrive after x_t before the elimination of x_t. As proved before, we maintain W arrivals in the structure and randomly choose one of them for removal. Thus, the probability to choose x_t for removal after i arrivals is $(1-\frac{1}{W})^{i-1}\frac{1}{W}$. The expected number of arrivals before removing x_t is: $\sum_{i=0}^{\infty} i(1-\frac{1}{W})^{i-1}\frac{1}{W} = \frac{1}{W}\frac{1}{(1-(1-\frac{1}{W}))^2} = W$. ☐

B. RAND-CELL Performance Analysis

As described, RAND-CELL consumes the same amount of memory as CELL, but has a higher overall error. In this section, we analyze this added error. When $t < W$, the sliding window is equivalent to the base algorithm CELL as described in Section IV. Accordingly, we focus on the case that $t \geq W$.

Denote by $\widehat{W} = \Sigma_{l\in levels} n_l E_l$ the window of the RAND-CELL algorithm where n_l is the number of elements in level l. Due to the approximate deletion of the oldest element, $\widehat{W}$ can be slightly different from W.

Given time point t, consider a flow f that belongs to level i and an arriving element x_t at time point t. Let I_A^t be the indicator of whether the arrival element x_t belongs to flow f and I_D^t be the indicator of whether the expired element belongs to flow f. Denote $N_{f,t}^W$ the number of elements with flow label f till time point t within the last W elements and by $\widehat{N_{f,t}^W}$ the estimation value of $N_{f,t}^W$. We have $\widehat{N_{f,t}^W} = E_i$, where E_i is the estimation value of level i ($E_i = A_\epsilon(i)$). This means that $\mathbb{E}(N_{f,t}^W - N_{f,t-1}^W) = \mathbb{E}(I_A^t) - \mathbb{E}(I_D^t)$.

If the flows are distributed i.i.d, we have[2]: $\mathbb{E}(I_D^t) = \frac{N_{f,t}^W}{W}$.

To calculate $\mathbb{E}(\widehat{N_{f,t}^W} - \widehat{N_{f,t-1}^W})$ we consider two cases:

- $x_t \in f$ and it causes f to climb to level $i+1$.
- The flow chosen for potential level degradation is f.

Denote $P_{CE}(f)$ the probability that flow f is chosen for potential degradation.

$$\begin{aligned}\mathbb{E}(\widehat{N_{f,t}^W} - \widehat{N_{f,t-1}^W}) &= P(I_A^t = 1) \cdot P(\textit{up level}) \cdot (E_{i+1} - E_i) \\ &\quad - P_{CE}(f) \cdot P(\textit{down level}) \cdot (E_i - E_{i-1}). \quad (7)\end{aligned}$$

According to CELL, flow f moves from level i to $i+1$ with probability $\frac{1}{E_{i+1}-E_i}$. Hence, $P(\textit{up level}) = \frac{1}{E_{i+1}-E_i}$. Similarly, $P(\textit{down level}) = \frac{1}{E_i-E_{i-1}}$ because we move flow f from its level i to the lower level $i-1$ with probability $\frac{1}{E_i-E_{i-1}}$. As we choose a flow for potential level degradation

[2]To be precise, we only need to assume that in each substream of slightly more than W elements, flows are distributed in an i.i.d. manner.

randomly from the last $\widehat{W}$ elements: $P_{CE}(f) = \frac{\widehat{N^W_{f,t}}}{\widehat{W}}$. Putting it all together, we get:

$$\mathbb{E}(\widehat{N^W_{f,t}} - \widehat{N^W_{f,t-1}}) = P(I^t_A = 1) \cdot \frac{1}{E_{i+1} - E_i} \cdot (E_{i+1} - E_i) - \frac{E_i}{\widehat{W}} \cdot (\frac{1}{E_i - E_{i-1}}) \cdot (E_i - E_{i-1}) = P(I^t_A = 1) - \frac{E_i}{\widehat{W}}. \quad (8)$$

$$\mathbb{E}(N^W_{f,t} - N^W_{f,t-1}) - \mathbb{E}(\widehat{N^W_{f,t}} - \widehat{N^W_{f,t-1}}) = \frac{E_i}{\widehat{W}} - \mathbb{E}(I^t_D). \quad (9)$$

Combining (V-B) with (9), to get:

$$\mathbb{E}(N^W_{f,t} - N^W_{f,t-1}) - \mathbb{E}(\widehat{N^W_{f,t}} - \widehat{N^W_{f,t-1}}) = \frac{E_i}{\widehat{W}} - \frac{N^W_{f,t}}{W}. \quad (10)$$

Using a Markov chain, we can find the estimated flow sizes. According to (10), when deriving $\widehat{N^W_{f,t}}$ from $\widehat{N^W_{f,t-1}}$, RAND-CELL introduces a small error of $\frac{E_i}{\widehat{W}} - \frac{N^W_{f,t}}{W}$, which is the difference between the estimated flow size and the accurate one. From [14], this difference is bounded by the given ϵ.

VI. The SHIFT-CELL Algorithm

SHIFT-CELL extends CELL for the sliding window model by performing a batch of estimators reductions once in every multiple of arrivals. In this section only, we assume that Query(f) operations are performed only for flows that arrive in the current window, i.e., among the last W items from the time of the query. This assumption is reasonable in many applications such as load balancing, traffic engineering, QoS enforcement, and many network security tasks, where the flow's frequency information is needed mainly in order to handle or classify an arriving item. Further, we do not provide a formal error bound for SHIFT-CELL, yet its empirical error on a real-world Internet trace is similar to RAND-CELL, as reported in Section VII-B. The strength of SHIFT-CELL comes from its improved performance compared to RAND-CELL, as we describe below. That is, SHIFT-CELL is very efficient at the cost of assurance on its theoretical error.

Ideally, aim to maintain the per-flow counting statistics only w.r.t. the W most recent elements in the stream. As mentioned before, to support this precisely, on every $ADD(x)$ operation we must also reduce the flow count of the least recent element in the data structure, which would require remembering the order in which elements from each flow have arrived (e.g., the approach taken by SWAMP [4]). Here, for efficiency, we approximate this by applying batch reductions on all flows.

SHIFT-CELL is illustrated in Figure 2 and is explained below. Specifically, in SHIFT-CELL, every time $ADD(x)$ is invoked, we increment a counter C. Next, whenever the counter C reaches the window size W, we subtract certain values, as defined shortly, from all estimators – a flow whose estimation becomes zero is eliminated (at least abstractly). The value of C is also reduced by the total number of reductions. Finally, regardless of whether estimator reductions were performed or not, the new element x is injected into the data structure in the same manner as in CELL.

The main challenge in this approach is to perform the periodic batch estimators reductions efficiently. The motivation

Algorithm 3 SHIFT-CELL Algorithm

```
   Initialization:
     initialize  Base = instance of the base algorithm CELL,
   Initialization: C ← 0
1: function ADD(x_i)
2:     C ← (C + 1)
3:     if C = W then
4:         Delete first level in Base
5:         Shifting levels in Base one level down
6:         Compute leftover value
7:         C ← W − leftover
8:     Base.Add(x_i)
9: function QUERY(f) return Base.Query(f)
```

for SHIFT-CELL is that after every W arrivals, we subtract $\approx W$ from the estimators of all flows, thereby approximately mimicking a sliding window. Further, "inside" the window, each subtraction only adds a small error to each flow. All this using the efficient mechanism described next, whose only memory overhead is the counter C, which takes $\log W$ bits, and as would be shown, is computationally also very efficient.

Periodic Batch Estimator Subtractions: We start by explaining the subtractions for the AM-based implementation of CELL and then for the HT representation. Denote by $node_i$ the data structure associated with level i. Each time counter C reaches W, we simply delete the first level (freeing $node_1$) and *shifting* every estimator one level down. That is, after the shift operation each $node_i$ becomes $node_{i-1}$ by adopting estimator E_{i-1} instead of E_i. This operation decreases the load of every level i from $n_i E_i$ to $n_i E_{i-1}$. This means that the total estimation reduction of level i is $n_i(E_i - E_{i-1})$. We denote by *leftover* the sum of estimation reductions of all the levels. Then, we update the counter C to be the difference between W and *leftover*, i.e., $W -$ *leftover*. The pseudo-code of the algorithm appears in Algorithm 3.

SHIFT-CELL incurs an expensive infrequent operation that goes over all the levels to calculate the value of the *leftover* value. Yet, as the number of levels is bounded by $O(\log W)$ (Observation1), this is also the computational cost of this periodic operation. Moreover, compared to RAND-CELL, the added cost of an $ADD(x)$ operation in SHIFT-CELL that does not result in a shift is only a single counter increment. In contrast, in RAND-CELL each $ADD(x)$ operation invokes the PRNG, which is computationally much more expensive.

Obviously, in the AM-based representation, shift is implemented trivially in a constant number of operations. For the fingerprint HT approach, we can have a *minLevel* counter, which is incremented on each shift operation. In this case, the level of a flow stored in the HT is interpreted as the value in the HT minus *minLevel*, with a negative result indicating a free entry. Such an implementation is computationally fast, but the level counters stored in the HT become unbounded.

VII. Evaluation

We developed a C++ prototype of all algorithms described in this work: CELL, RAND-CELL and SHIFT-CELL. We considered 3 implementations of CELL: an approximate membership representation using TinySet (aka AM), a hash table representation using TinyTable (aka HT_TinyTable), and a hash table representation using Cuckoo [35] (aka HT_Cuckoo)

and compared them with two state-of-the-art counter estimation algorithms, CEDAR [51] and ICE-Buckets [14] (aka ICE) and with the naive solution, which is using a perfect hash table and accurate counters as a general baseline. For the sliding window model, we compared RAND-CELL and SHIFT-CELL with SWAMP [4], a state-of-the-art sliding window algorithm that solves accurate per-flow counting. We have implemented ICE-Buckets [14] and SWAMP [4] in C++ ourselves since their authors' code is in Python and Java.

Our evaluation utilizes the following datasets:

- **CAIDA** - The CAIDA Anonymized Internet Trace 2016 [21]. The dataset collected during 2016 from the backbone router 'equinix-chicago' which contains 88 million packets from 1.65M unique flows.
- **Synthetic datasets** - Generated datasets that follow the Zipf distribution ($p(x) = \frac{x^{-a}}{\zeta(a)}$, $a = 0$).

We used IP 5-tuples as flow ids. The evaluation was performed on an Intel(R) 3.20GHz Xeon(R) CPU E5-2667 v4 running Linux with kernel 4.4.0-71.

A. Memory Consumption Comparison

In this section, we compare the space consumption of our algorithms as well as CEDAR, ICE-Buckets, and the naive solution for a given ϵ and δ values. Figure 3 shows the space comparison as a function of the accuracy guarantee ϵ while the configured error probability $\delta = 0.01$. For ICE-Buckets and CEDAR, we preprocessed the dataset to obtain the maximal number of the unique flows. The memory for each algorithm was computed by counting the number of bits in the implementation parameters given during runtime.

a) Effect of knowing the number of unique flows on memory consumption: Figure 3a shows the memory consumption with prior knowledge of the number of the flows (the trace contains about 88 million packets and the number of unique flows is $1,660,000$). When considering trace statistics for CAIDA [2], for many traces, the number of unique flows is $10 - 50$ times lower than the stream size.

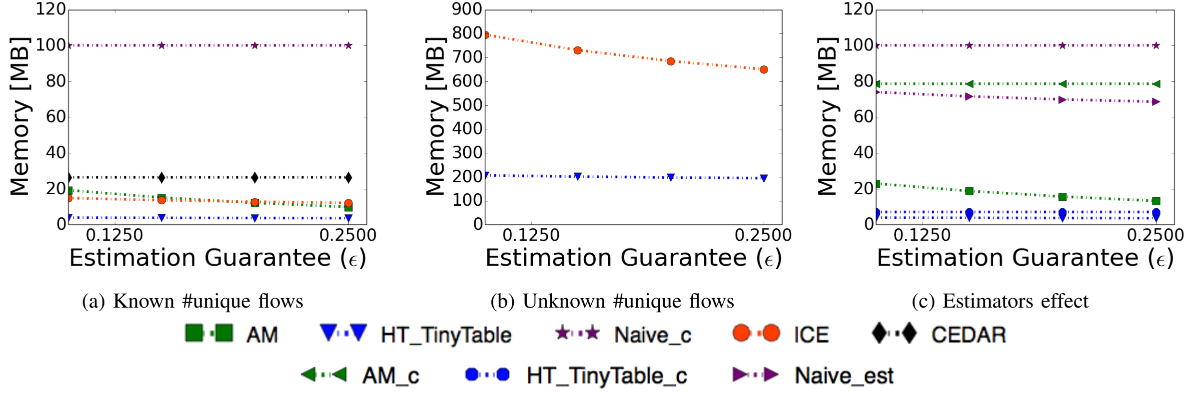
The naive solution is not affected by varying ϵ values since it returns accurate answers. CEDAR maintains pointers from each flow to its counter, so its space consumption is generally the largest. CEDAR and ICE-Buckets are more compact than AM for very small values of ϵ but consume more space than HT_TinyTable for all ϵ values. For the AM algorithms, as ϵ increases, the overall number of levels decreases resulting in better space consumption. Yet, HT_TinyTable is the most efficient among the algorithms that solve the problem by at least 2X due to its independent self-adjusting counter sizes.

Indeed the AM-based representation wastes additional memory compared to HT_TinyTable, but its advantage comes from the simple implementation for RAND-CELL and SHIFT-CELL as described before. As mentioned in [51] and [14], CEDAR and ICE-Buckets must know a-priory the number of unique flows to allocate compact data structures; otherwise, they use the stream size as a bound of the number of the unique flows, which cause a significant memory overhead. Figure 3b shows the memory overhead of CELL and ICE-Buckets for the same 88 million packets when the number of unique flows is unknown. As seen, having no a-priory knowledge of the number of unique flows increases the memory overhead gap between CELL and ICE-Buckets. The main extra overhead for this case comes from the fact the tables need to be sized for the worst case in which each item is from a unique flow.

b) Effect of using estimators on memory consumption: To demonstrate the motivation for using estimators instead of accurate counters, we explore the memory consumption for the version of each algorithm (AM, HT_TinyTable, and Naive) that uses estimators compared with the same algorithm using accurate counters. Figure 3c shows this memory consumption as a function of ϵ when the number of unique flows is known. As shown in Figure 3c, when the implementation employs accurate counters its results are not affected by the value of ϵ since the algorithm supplies accurate answers. As shown, using estimators instead of accurate counters improves the memory overhead in all algorithms. The AM algorithm with estimators consumes $71\% - 84\%$ less space, while HT_TinyTable with estimators consumes 50% less. Even if we apply estimators to the naive solution, it saves 30% of the memory consumption. The reason AM is the most affected is that its memory consumption is the most severely dependent on the number of layers, which is exactly what estimators reduce.

c) Effect of false positive ratio δ on memory consumption: Figure 4a shows the effect of the false positive ratio on CELL's memory consumption with the three implementations: HT_TinyTable, HT_Cuckoo and AM when ϵ is fixed to 0.1. In this comparison, we consider the case of an unknown number of unique flows. All algorithms consume more memory when the false-positive ratio is larger as this means more bits for the fingerprint in the HT representation and larger approximate memberships in the AM representation. As expected, AM consumes more memory compared to HT, while HT_TinyTable is the lightest algorithm under the same configurations.

d) Memory Consumption over Sliding Window: We now compare the space consumption of our sliding window algorithms, RAND-CELL and SHIFT-CELL with SWAMP [4] (configured with $\delta = 0.01$) as a function of the window size, when ϵ is fixed to 0.1. As described in sections V and VI, RAND-CELL and SHIFT-CELL adapt CELL to the sliding window model without adding any memory overhead at the cost of a larger error (SHIFT-CELL adds a counter, but its cost is very small). So the memory consumption of the two algorithms as a function of the window size is the same. In Figure 4b we compare the two implementations of CELL, AM and HT_TinyTable, when adapted to the sliding window model with the state-of-the-art SWAMP without any further knowledge about the number of the unique flows. All algorithms consume more memory when the window size is larger, as this means maintaining more elements in their data structures. SWAMP stores all window elements' fingerprints in a cyclic buffer plus a TinyTable of the fingerprints. Moreover, the counters in the TinyTable are exact values (not estimators). These promise accurate answers with probability $1-\delta$ but lead to significant memory usage. As seen, SWAMP consumes 4

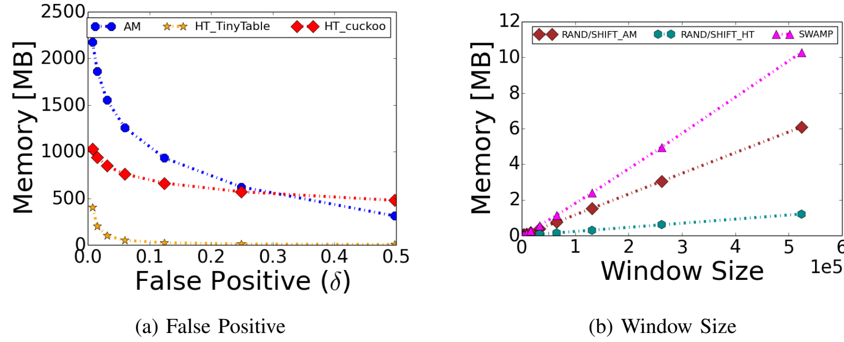


(a) Known #unique flows

(b) Unknown #unique flows

(c) Estimators effect

Fig. 3: Space as a function of the accuracy guarantee (ϵ): (a) the number of the unique flows is known, (b) the number of the unique flows is unknown, (c) the effect of using estimators on memory – we compare space consuming of each algorithm using estimators vs. real value. (*_c indicates exact counters variants, *_est indicates estimators).



(a) False Positive

(b) Window Size

Fig. 4: (a) Space as a function of the false positive ratio (δ) when the number of the unique flows is unknown (b) Space as a function of the window size in the sliding window model without any knowledge of the number of the unique flow.

times more memory for large window sizes (2^{17}) than our compact implementation (HT_TinyTable).

B. Error (RMSRE) Comparison

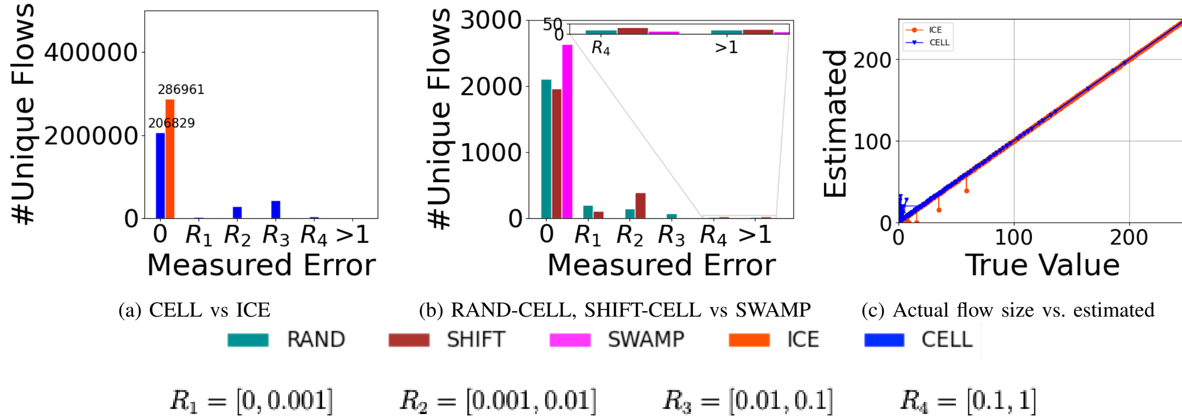
We now explore the empirical error of our algorithms and compare them to previous works. We considered CELL's HT_TinyTable implementation as it is the most compact representation of CELL. We configured our algorithms with $\epsilon = 0.1$ and $\delta = 2^{-9} \approx 0.00194$. Specially, we compared CELL with ICE-Buckets, and for the sliding window model, we compared RAND-CELL and SHIFT-CELL with SWAMP. To configure ICE-Buckets, we preprocessed the dataset to obtain the maximal number of the unique flows and the trace size (which is a requirement as mentioned before), and set ϵ to 0.1, while SWAMP is configured with the same δ (2^{-9}). Figure 5 compares our algorithms with ICE-Buckets and SWAMP over a synthetic dataset of 5 million packets. Figure 6 shows the empirical RMSRE for different window sizes.

Figure 5a shows a histogram of unique flows number per range of the measured error when $\epsilon = 0.1$ and $\delta = 2^{-9} \approx 0.00194$. As seen, both CELL and ICE-Buckets provide **accurate** answers for the vast majority of flows. ICE-Buckets is more accurate than CELL as it is a deterministic algorithm while CELL has an error probability. However, ICE-Buckets must know a-priory the number of the unique flows; otherwise, it assumes that the number of unique flows is the stream size, which causes a significant memory overhead as seen in Figure 3b. Moreover, CELL is more compact than ICE-Buckets both when the number of the unique flows is known and when it is unknown, as shown in Figures3a and 3b.

The high accuracy of CELL is also exhibited in Figure 5c. It shows the estimated flow size as a function of the true value. CELL's estimated values that are very close to the true ones.

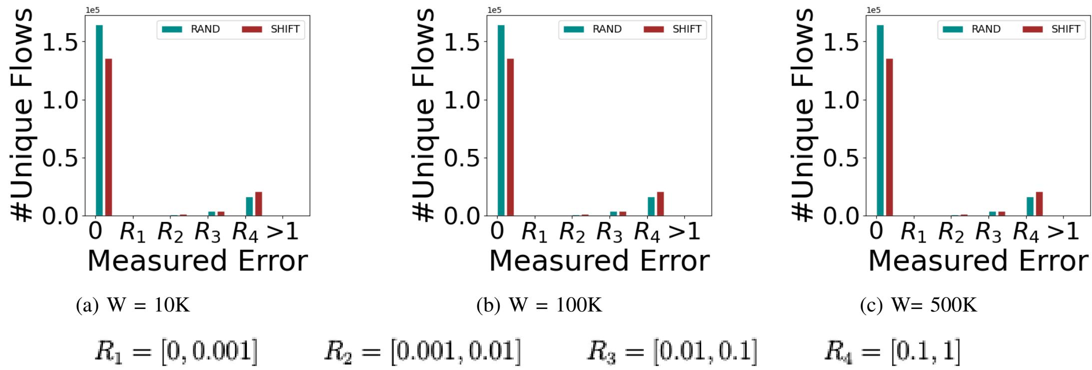
Figure 5b compares RAND-CELL and SHIFT-CELL with SWAMP. Since SWAMP provides accurate estimations, it is more accurate than RAND-CELL and SHIFT-CELL. But, it promises accurate estimations only with some probability (configured with $\delta = 0.00194$). As seen, for high measured error, SWAMP and our algorithms are very close (the zoomed figure shows this). Alas, SWAMP consumes much more space as it maintains the whole window size in memory, exposing an accuracy to memory consumption trade-off between SWAMP and our algorithms (RAND-CELL and SHIFT-CELL).

In conclusion, ICE-Buckets and SWAMP are slightly more accurate than our algorithms but this comes at the cost of more memory consumption (as seen in Figure 3). Notice that



(a) CELL vs ICE (b) RAND-CELL, SHIFT-CELL vs SWAMP (c) Actual flow size vs. estimated

Fig. 5: Using synthetic dataset (a) Number of unique flows over measured error (overall error) for CELL and ICE-Buckets when $\epsilon = 0.1$, $\delta = 0.00194$ and window size $= 2^{16}$. (b) Number of unique flows over measured error (overall error) for RAND-CELL and SHIFT-CELL when $\epsilon = 0.1$ and $\delta = 0.00194$ (c) Actual flow size vs. estimated flow size for CELL when $\epsilon = 0.1$ and $\delta = 0.00194$.



(a) W = 10K (b) W = 100K (c) W= 500K

Fig. 6: Unique flows over measured (overall) error for SHIFT-CELL and RAND-CELL configured with $\epsilon = 0.1$ and $\delta = 0.031$ using Chicago dataset of size 1M packets: (a) window size 10K (b) window size 100K (c) window size 500K

the observed errors are lower than the user-selected value.

Figures 6 measures the empirical RMSRE for RAND-CELL and SHIFT-CELL in the HT_TinyTable implementation and reported them as histograms. In this experiment, after every W arrivals, we compared the estimates of each flow in the window with the accurate figure. As expected, the gap between both algorithms gets closer when the window size is larger.

VIII. CONCLUSION

We have introduced CELL, a novel counter estimation algorithm that minimizes the memory overhead while guaranteeing a given error of ϵ with probability $1-\delta$. CELL maps the flows to levels according to their frequencies and uses estimators to answer per-flow counting according to the flow's level. We studied two possible representations of CELL, a Hash Table (HT) representation and an Approximate Membership (AM), and analyzed the memory consumption and the value of δ obtained by each of them for a given ϵ. We have compared CELL to CEDAR [51] and ICE-Buckets [14] in terms of memory consumption over real Internet traces. We showed that the HT variants of CELL consume less memory than both CEDAR and ICE-Buckets for the same accuracy error.

We extended CELL for the sliding window model and we have presented two extensions of CELL to the sliding window model, RAND-CELL and SHIFT-CELL. Both our sliding window algorithms were shown to be more space-efficient than SWAMP [4], the previously best-known solution for per-flow accurate counting over a sliding window (also with probability $1-\delta$). By comparing to SWAMP, we showed the significant memory reduction that can be obtained in the sliding window model by allowing a small approximation error. Among our two algorithms, RAND-CELL is better when a formal error guarantee is required, while SHIFT-CELL is computationally lighter. All code is available online [1].

Acknowledgments: We would like to thank our shepherd Dan Li and the anonymous reviewers for helping improve our paper. The work was partially funded by the Technion HPI research school and ISF grant #1505/16.

REFERENCES

[1] Refernce removed due to anonymity - an open source project.

[2] Trace Statistics for CAIDA https://www.caida.org/data/passive/trace_stats.

[3] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 286–296. ACM, 2004.

[4] Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. Pay for a Sliding Bloom Filter and Get Counting, Distinct Elements, and Entropy for Free. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 2204–2212, 2018.

[5] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized Admission Policy for Efficient Top-k, Frequency, and Volume Estimation. *IEEE/ACM Trans. Netw.*, 27(4):1432–1445, 2019.

[6] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *INFOCOM*, pages 1–9, 2016.

[7] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An Improved Construction for Counting Bloom Filters. In *14th Annual European Symposium on Algorithms, LNCS*, ESA, pages 684–695. Springer, 2006.

[9] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. Bloom Filters via d-Left Hashing and Dynamic Bit Reassignment. In *Allerton Conf. on Communication, Control and Computing*, 2006.

[10] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding Frequent Items in Data Streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[11] Graham Cormode and Shan Muthukrishnan. An Improved Data Stream Summary: the Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[12] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[13] G. Einziger and R. Friedman. TinySet — An Access Efficient Self Adjusting Bloom Filter Construction. *IEEE/ACM Transactions on Networking*, 25(4):2295–2307, 2017.

[14] Gil Einziger, Benny Fellman, Roy Friedman, and Yaron Kassner. ICE Buckets: Improved Counter Estimation for Network Measurement. *IEEE/ACM Transactions on Networking*, 26(3):1165–1178, 2018.

[15] Gil Einziger and Roy Friedman. Counting With Tinytable: Every Bit Counts! *IEEE Access*, 7:166292–166309, 2019.

[16] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and Elastic DDOS Defense. In *24th USENIX Security Symposium (USENIX Security)*, pages 817–832, 2015.

[17] Domenico Ficara, Andrea Di Pietro, Stefano Giordano, Gregorio Procissi, and Fabio Vitucci. Enhancing Counting Bloom Filters Through Huffman-Coded Multilayer Structures. *IEEE/ACM Trans. Netw.*, 18(6):1977–1987, 2010.

[18] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges. *computers & security*, 28(1-2):18–28, 2009.

[19] Michael T Goodrich and Michael Mitzenmacher. Invertible Bloom Lookup Tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799, 2011.

[20] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. Sliding sketches: A framework using time zones for data stream processing in sliding windows. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1015–1025, 2020.

[21] Paul Hick. CAIDA Anonymized Internet Trace, equinix-chicago, 2016.

[22] Chengchen Hu and Bin Liu. Self-Tuning the Parameter of Adaptive Non-Linear Sampling Method for Flow Statistics. In *2009 International Conference on Computational Science and Engineering*, volume 1, pages 16–21. IEEE, 2009.

[23] Chengchen Hu, Bin Liu, Hongbo Zhao, Kai Chen, Yan Chen, Chunming Wu, and Yu Cheng. DISCO: Memory Efficient and Accurate Flow Statistics for Network Measurement. In *IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 665–674, 2010.

[24] Nan Hua, Bill Lin, Jun Xu, and Haiquan Zhao. BRICK: A Novel Exact Active Statistics Counter Architecture. In *Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 89–98, 2008.

[25] Kun Huang, Jie Zhang, Dafang Zhang, Gaogang Xie, K. Salamatian, AX. Liu, and Wei Li. A Multi-Partitioning Approach to Building Fast and Accurate Counting Bloom Filters. In *IEEE IPDPS*, 2013.

[26] Regant YS Hung, Lap-Kei Lee, and Hing-Fung Ting. Finding frequent items over sliding windows with constant update time. *Information Processing Letters*, 110(7):257–260, 2010.

[27] Yossi Kanizo, David Hay, and Isaac Keslassy. Access-Efficient Balanced Bloom Filters. *Computer communications*, 36(4):373–385, 2013.

[28] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions Database Systems*, 2003.

[29] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An Architecture for Global-Scale Persistent Storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[30] Lap-Kei Lee and HF Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 290–297. ACM, 2006.

[31] Yeonhee Lee and Youngseok Lee. Toward Scalable Internet Traffic Measurement and Analysis with Hadoop. *ACM SIGCOMM Computer Communication Review*, 43(1):5–13, 2012.

[32] Lichun Li, Bingqiang Wang, and Julong Lan. A Variable Length Counting Bloom Filter. In *2nd Int. Conf. on Computer Engineering and Technology (ICCET)*, volume 3, 2010.

[33] Wei Li, Kun Huang, Dafang Zhang, and Zheng Qin. Accurate Counting Bloom Filters for Large-Scale Data Processing. *Mathematical Problems in Engineering*, 2013.

[34] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter Braids: a Novel Counter Architecture for Per-Flow Measurement. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):121–132, 2008.

[35] Steve Lumetta and Michael Mitzenmacher. Using the Power of Two Choices to Improve Bloom Filters. *Internet Mathematics*, 4(1):17–33, 2007.

[36] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient Computation of Frequent and top-K Elements in Data Streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

[37] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[38] Jayadev Misra and David Gries. Finding Repeated Elements. Technical report, 1982.

[39] Robert Morris. Counting Large Numbers of Events in Small Registers. *Commun. ACM*, 21(10):840–842, 1978.

[40] Robert Morris. Counting Large Numbers of Events in Small Registers. *Communications of the ACM*, 21(10):840–842, 1978.

[41] K. Pagiamtzis and A. Sheikholeslami. Content-Addressable Memory (CAM) Circuits and Architectures: a Tutorial and Survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, 2006.

[42] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketching distributed sliding-window data streams. *The VLDB Journal—The International Journal on Very Large Data Bases*, 24(3):345–368, 2015.

[43] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, Hash-and Space-Efficient Bloom Filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[44] Yan Qiao, Tao Li, and Shigang Chen. One Memory Access Bloom Filters and Their Generalization. In *Proceedings IEEE INFOCOM*, pages 1745–1753, 2011.

[45] Sriram Ramabhadran and George Varghese. Efficient Implementation of a Statistics Counter Architecture. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 261–271, 2003.

[46] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. The Variable-Increment Counting Bloom Filter. In *IEEE INFOCOM*, pages 1880–1888, 2012.

[47] Devavrat Shah, Sundar Iyer, B Prahhakar, and Nick McKeown. Maintaining Statistics Counters in Router Line Cards. *IEEE Micro*, 22(1):76–81, 2002.

[48] Anshumali Shrivastava, Arnd Christian Konig, and Mikhail Bilenko. Time adaptive sketches (ada-sketches) for summarizing data streams. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1417–1432, 2016.

[49] Rade Stanojevic. Small Active Counters. In *26th IEEE International Conference on Computer Communications (INFOCOM)*, pages 2153–2161, 2007.

[50] Erez Tsidon, Iddo Hanniel, and Isaac Keslassy. Estimators Also Need Shared Values to Grow Together. In *IEEE INFOCOM*, pages 1889–1897, 2012.

[51] Erez Tsidon, Iddo Hanniel, and Isaac Keslassy. Estimators Also Need Shared Values to Grow Together. In *Proceedings of IEEE INFOCOM*, pages 1889–1897, 2012.

[52] Sen Yang, Bill Lin, and Jun Xu. Safe Randomized Load-Balanced Switching By Diffusing Extra Loads. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):1–37, 2017.

[53] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[54] Qi Zhao, Jun Xu, and Zhen Liu. Design of a Novel Statistics Counter Architecture with Optimal Space and Time Efficiency. In *Proc. of the joint ACM Int. Conf. on Measurement and Modeling of Computer Systems*, SIGMETRICS, pages 323–334, 2006.