Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications

Yin Zhang[⋆] Sumeet Singh[§] Subhabrata Sen[⋆] Nick Duffield[⋆] Carsten Lund[⋆]

AT&T Labs – Research, Florham Park, NJ 07932, USA[⋆]
CSE Department, University of California, San Diego, CA 92040, USA[§]

{yzhang,sen,duffield,lund}@research.att.com susingh@cs.ucsd.edu

ABSTRACT

In traffic monitoring, accounting, and network anomaly detection, it is often important to be able to detect high-volume traffic clusters in near real-time. Such heavy-hitter traffic clusters are often hierarchical (*i.e.*, they may occur at different aggregation levels like ranges of IP addresses) and possibly multidimensional (*i.e.*, they may involve the combination of different IP header fields like IP addresses, port numbers, and protocol). Without prior knowledge about the precise structures of such traffic clusters, a naive approach would require the monitoring system to examine all possible combinations of aggregates in order to detect the heavy hitters, which can be prohibitive in terms of computation resources.

In this paper, we focus on online identification of 1-dimensional and 2-dimensional hierarchical heavy hitters (HHHs), arguably the two most important scenarios in traffic analysis. We show that the problem of HHH detection can be transformed to one of dynamic packet classification by taking a top-down approach and adaptively creating new rules to match HHHs. We then adapt several existing static packet classification algorithms to support dynamic packet classification. The resulting HHH detection algorithms have much lower worst-case update costs than existing algorithms and can provide tunable deterministic accuracy guarantees. As an application of these algorithms, we also propose robust techniques to detect changes among heavy-hitter traffic clusters. Our techniques can accommodate variability due to sampling that is increasingly used in network measurement. Evaluation based on real Internet traces collected at a Tier-1 ISP suggests that these techniques are remarkably accurate and efficient.

Categories and Subject Descriptors

C.2.3 [**Computer-Communications Networks**]: Network Operations—*Network Monitoring, Network Management*

General Terms

Measurement, Algorithms

Keywords

Network Anomaly Detection, Data Stream Computation, Hierarchical Heavy Hitters, Change Detection, Packet Classification

1. INTRODUCTION

1.1 Motivation and background

The Internet has emerged as a critical communication infrastructure, carrying traffic for a wide range of important scientific, business and consumer applications. Network service providers and enterprise network operators need the ability to detect anomalous events in the network, for network management and monitoring, reliability, security and performance reasons. While some traffic anomalies are relatively benign and tolerable, others can be symptomatic of potentially serious problems such as performance bottlenecks due to flash crowds [24], network element failures, malicious activities such as denial of service attacks (DoS) [23], and worm propagation [28]. It is therefore very important to be able to detect traffic anomalies accurately and in near real-time, to enable timely initiation of appropriate mitigation steps. This paper focuses on streaming techniques for enabling accurate, near real-time detection of anomalies in IP network traffic data.

A major challenge for anomaly detection is that traffic anomalies often have very complicated structures: they are often hierarchical (*i.e.*, they may occur at arbitrary aggregation levels like ranges of IP addresses and port numbers) and sometimes also multidimensional (*i.e.*, they can only be exposed when we examine traffic with specific combinations of IP address ranges, port numbers, and protocol). In order to identify such multidimensional hierarchical traffic anomalies, a naive approach would require the monitoring system to examine all possible combinations of aggregates, which can be prohibitive even for just two dimensions. Another challenge is the need to process massive streams of traffic data online and in near real-time. Given today's traffic volume and link speeds, the input data stream can easily contain millions or more of concurrent flows, so it is often infeasible or too expensive to maintain per-flow state.

1.2 Heavy hitters, aggregation and hierarchies

A very useful concept in identifying dominant or unusual traffic patterns is that of hierarchical heavy hitters (HHHs) [11]. A *heavy hitter* is an entity which accounts for at least a specified proportion of the total activity measured in terms of number of packets, bytes, connections etc. A heavy hitter could correspond to an individual flow or connection. It could also be an aggregation of multiple flows/connections that share some common property, but which themselves may not be heavy hitters.

Of particular interest to our application is the notion of hierarchical aggregation. IP addresses can be organized into a hierarchy according to prefix. The challenge for hierarchical aggregation is to efficiently compute the total activity of all traffic matching relevant prefixes. A *hierarchical heavy hitter* is a hierarchical aggregate that accounts for some specified proportion of the total activity.

Aggregations can be defined on one or more dimensions, *e.g.*, source IP address, destination IP address, source port, destination port, and protocol fields for IP flows. Correspondingly, in this paper we will be concerned with *multidimensional hierarchical heavy hitters*, *i.e.*, multidimensional sets of hierarchical aggregates that account for some specified proportion of the total activity.

1.3 Contribution and approach

The main contribution of this paper is the development of several efficient streaming algorithms for detecting multidimensional hierarchical heavy hitters from massive data streams with a large number of flows. The common component of these algorithms is an adaptive data structure that carries a synopsis of the traffic in the form of a set of estimated hierarchical aggregates of traffic activity. The data structure is adapted to the offered traffic in that each aggregate contains no more than a given proportion of the total activity (with possible exception for those aggregates that are not further divisible).

These algorithms have much lower worst-case update costs than existing algorithms, and provide data independent deterministic accuracy guarantees. By adjusting the threshold proportion for detection, the level of detail reported can be traded off against the computation time.

A key theoretical contribution that enables our work is that we establish the close connection between multidimensional hierarchical heavy hitter detection and packet classification, two important problems often studied separately in the literature. In packet classification one maps packets onto a given set of fixed prefixes. Our problem is more challenging in that the set of prefixes (corresponding to the heavy-hitter traffic clusters) is dynamic, adapting to the set of IP addresses presented by the traffic and the relative activity on each of the prefixes. In fact, all our algorithms have static counterparts in the packet classification world (*e.g.*, [31, 33]).

Our original motivation for this work was network anomaly detection. Change detection, an important component in anomaly detection, involves detecting traffic anomalies by deriving a model of normal behavior based on the past traffic history and looking for significant changes in short-term behavior (on the order of minutes to hours) that are inconsistent with the model. In the present context, this requires detecting changes across time in the activity associated with the heavy hitters. As an application of our method, we describe how standard change detection techniques can be adapted for robust use with the activity time series of hierarchical heavy hitters generated from the measured traffic. Evaluation based on real Internet traces collected at a Tier-1 ISP suggests that these techniques are remarkably accurate and efficient.

An important challenge to change detection stems from the fact that usage measurements are increasingly sampled. For instance, for NetFlow data, there are typically 2 levels of sampling: (i) packet sampling at the routers during the formation of NetFlow records [10], and (ii) smart sampling [16, 15] of the NetFlow records within the measurement infrastructure. Our techniques accommodate the inherent sampling variability within our predictive scheme. Specifically, we can set alarm thresholds in order to keep the false positive rate due to sampling variability within acceptable limits.

1.4 Related work

There is considerable literature in the area of statistical anomaly detection. Change detection has been extensively studied in the context of time series forecasting and outlier analysis [34, 9]. The standard techniques include simple smoothing techniques (*e.g.*, exponential averaging), the more general Box-Jenkins ARIMA modeling [6, 7, 1], and wavelet-based methods [5, 4]. Prior works have applied these techniques to network fault detection (*e.g.*, [22, 25, 35, 19]) and intrusion detection (*e.g.*, [8]). Barford *et al.* recently provided a good characterization of different types of anomalies [5] and proposed wavelet-based methods for change detection [4].

Existing works on heavy hitter detection lack the multidimensional adaptive hierarchical drill-down capability that our deterministic techniques offer. Existing change detection techniques typically can only handle a relatively small number of time series. Recent efforts use probabilistic summarization techniques like sketches to avoid per-flow state, for scalable heavy hitter detection [13, 14, 17] and change detection [26]. [11] presents both deterministic and sketch-based probabilistic online algorithms for hierarchical heavy hitter detection in one dimension. [18] presents effective techniques for offline computation of multidimensional heavy hitters. Recently, Cormode *et al.* [12] proposed an algorithm for multidimensional heavy hitter detection, which is the closest in spirit to our work. We will discuss their algorithm further at the end of Section 3.1.

The remainder of the paper is organized as follows: Section 2 formally presents the multidimensional HHH detection and change detection problems. Section 3 provides detailed descriptions of our proposed multidimensional HHH detection algorithms, and Section 4 describes our proposed techniques for change detection for HHH clusters. Section 5 outlines our evaluation methodology, and Section 6 presents evaluation results for our HHH detection and change detection algorithms. Finally, Section 7 concludes the paper.

2. PROBLEM SPECIFICATION

In this section, we formally define the notion of multidimensional hierarchical heavy hitters and introduce the heavy hitter detection problem.

We adopt the Cash Register Model [29] to describe the streaming data. Let $\mathcal{I} = \alpha_1, \alpha_2, \cdots$, be an input stream of items that arrives sequentially. Each item $\alpha_i = (k_i, u_i)$ consists of a key k_i, and a *positive* update $u_i \in \mathbb{R}$. Associated with each key k is a time varying signal $A[k]$. The arrival of each new data item (k_i, u_i) causes the underlying signal $A[k_i]$ to be updated: $A[k_i] += u_i$.

Below we first review the definition of Heavy Hitter and Hierarchical Heavy Hitters.

DEFINITION 1 (HEAVY HITTER). *Given an input stream $I = \{(k_i, u_i)\}$ with total sum $SUM = \sum_i u_i$ and a threshold ϕ ($0 \leq \phi \leq 1$), a Heavy Hitter (HH) is a key k whose associated total value in I is no smaller than ϕSUM. More precisely, let $v_k = \sum_{i:k_i=k} u_i$ denote the total value associated with each key k in I. The set of Heavy Hitters is defined as $\{k | v_k \geq \phi SUM\}$.*

We define the *heavy hitter* problem as the problem of finding all heavy hitters, and their associated values, in a data stream. For instance, if we use the destination IP address as the key, and the byte count as the value, then the corresponding HH problem is to find all destination IP addresses that account for at least a proportion ϕ of the total traffic.

DEFINITION 2 (HIERARCHICAL HEAVY HITTER). *Let $I = \{(k_i, u_i)\}$ be an input stream whose keys k_i are drawn from a hierarchical domain D of height h. For any prefix p of the domain hierarchy, let $elem(D, p)$ be the set of elements in D that are descendents of p. Let $V(D, p) = \sum_k v_k : k \in elem(D, p)$ denote the total value associated with any given prefix p. The set of Hierarchical Heavy Hitters (HHH) is defined as $\{p | V(D, p) \geq \phi SUM\}$.*

We define the *hierarchical heavy hitter* problem as the problem of finding all hierarchical heavy hitters, and their associated values, in a data stream. If we use the destination IP address to define the hierarchical domain, then the corresponding HHH problem not only wants to find destination IP addresses but also all those destination prefixes that account for at least a proportion ϕ of the total traffic.

Note that our definition of HHH is *different* from that of [11, 18]. Specifically, we would like to find all the HH prefixes, whereas [11, 18] returns a prefix p only if its traffic remains above ϕSUM even after excluding all traffic from HH prefixes that are descendents of p. All our algorithms can be adapted to use this more strict definition of HHH. We choose to use a simpler definition as part of our goal of HHH detection is to perform change detection on HHHs. If we do not output all the heavy hitter prefixes, then we can easily miss those big changes buried inside the prefixes that were not tracked (under the more strict definition).

We can generalize the definition of HHH to multiple dimensions:

DEFINITION 3 (MULTIDIMENSIONAL HHH). *Let $D = D_1 \times \cdots \times D_n$ be the Cartesian product of n hierarchical domains D_j of height h_j $(j = 1, 2, \cdots, n)$. For any $p = (p_1, p_2, \cdots, p_n) \in D$, let $elem(D, p) = elem(D_1, p_1) \times \cdots \times elem(D_n, p_n)$. Given an input stream $I = \{(k_i, u_i)\}$, where k_i is drawn from D, let $V(D, p) = \sum_k v_k : k \in elem(D, p)$. The set of Multidimensional Hierarchical Heavy Hitters is defined as $\{p | V(D, p) \geq \phi SUM\}$.*

For simplicity, we also refer to a multidimensional hierarchical heavy hitter as a *HHH cluster* in the rest of the paper.

The *multidimensional hierarchical heavy hitter* problem is defined as the problem of finding all multidimensional hierarchical heavy hitters, and their associated values, in a data stream. As an example, we can define D based on source and destination IP addresses. The corresponding 2-dimensional HHH problem is to find all those source-destination prefix combinations $< p_1, p_2 >$ that account for at least a proportion ϕ of the total traffic.

Once the multidimensional hierarchical heavy hitters have been detected in each time interval, we then need to track their values across time to detect significant changes, which may indicate potential anomalies. We refer to this as the *change detection* problem.

Our goal in this paper is to develop efficient and accurate streaming algorithms for detecting multidimensional hierarchical heavy hitters and significant changes in massive data streams that are typical of today's IP traffic.

3. MULTIDIMENSIONAL HHH DETECTION

To recall, our goal is to identify all possible keys (in the context of network traffic a key can be made up of fields in the packet header) that have a volume associated with them that is greater than the heavy-hitter *detection threshold* at the end of the time interval. A key may be associated with very large ranges. For example in the case of IP prefixes the range is: $[0, 2^{32})$. Also the key may be a combination of one or more fields, which can result in significant increase in the complexity of the problem. Clearly monitoring all possible keys in the entire range can be prohibitive (especially in the multidimensional context where we would have to consider a cross-product of all the individual ranges).

Our solution to this problem entails building an adaptive data structure that dynamically adjusts the granularity of the monitoring process to ensure that the particular keys that are heavy-hitters (or more likely to be heavy-hitters) are correctly identified without wasting a lot of resources (in terms of time and space) for keys that are not heavy-hitters. In the 1-dimensional case, our data structure resembles a decision tree that dynamically drills down and starts monitoring a node (that is associated with a key) closely only when its direct ancestor becomes sufficiently large. In the 2-dimensional case, our data structure provides similar dynamic drill-down capability.

There are two key parameters that we will use throughout the rest of the paper: ϕ and ϵ. Given the total sum SUM, ϕSUM is the threshold for a cluster to qualify as a heavy hitter; ϵSUM specifies the maximum amount of inaccuracy that we are willing to tolerate in the estimates generated by our algorithms.

To guide the building process of the summary data structure, we use a threshold, which we call the *split threshold* (T_{split}), to make local decisions at each step. It is used to make a decision as to when the range of keys under consideration should be looked at in a finer grain. T_{split} is chosen to ensure that the maximum amount of traffic we miss during the dynamic drill-down is at most ϵSUM for any cluster. The actual choice of T_{split} depends on the algorithm. *For now we assume that SUM is a pre-specified constant.* Later in Section 3.6, we will introduce a simple technique that allows us to specify T_{split} in terms of the actual total sum in a given time interval.

To exemplify the algorithms described in this section, we consider the source and the destination IP fields as the two dimensions for HHH detection. We also use what we call the *volume*, the number of bytes of traffic, associated with a given key, as the metric that we would like to use for detecting heavy-hitters. The metric as well as the fields to be considered for the dimensions may be changed based on the application requirements.

We start by considering a simple baseline solution to the HHH detection problem followed by adaptive algorithms for 1-dimensional and 2-dimensional HHH detection, arguably the two most important scenarios for traffic analysis. We conclude this section with a discussion on how our algorithms can be used as building blocks for general n-dimensional HHH detection.

3.1 Baseline solution

Below we describe a relatively straightforward, albeit inefficient, solution to the n-dimensional HHH detection problem. The scheme transforms the problem to essentially multiple (non-hierarchical) HH detection problems, one for each distinct combination of prefix length values across all the dimensions of the original key space. For an n-dimensional keyspace with a hierarchy of height h_i in the i-th dimension, there are $\Pi_{i=1}^{n}(h_i + 1)$ non-hierarchical HH detection problems, which have to be solved in tandem. Such a brute force approach will need to update the data structure for all possible combinations of prefix lengths. So the per-item update time is proportional to $\Pi_{i=1}^{n}(h_i + 1)$.

We use the above approach as a baseline for evaluating the multidimensional HHH detection algorithms proposed later in this section. We use the following two baseline variants that differ in the specific HH detection algorithm used. In the interest of space, we only provide a high level summary of the HH detection algorithms; readers are referred to [14, 27] for detailed descriptions.

Baseline variant 1: *Sketch-based* solution (sk), which uses *sketch*-based probabilistic HH detection. Count-Min sketch [14] is a probabilistic summary data structure based on random projections (see [29] for a good overview of sketch and specific sketch operations). Let $[m]$ denote set $\{0, 1, \cdots, m-1\}$. A sketch S consists of a $H \times K$ table of registers: $T_S[i, j]$ $(i \in [H], j \in [K])$. Each row $T_S[i, \cdot]$ $(i \in [H])$ is associated with a hash function h_i that maps the original key space to $[K]$. We can view the data structure as an array of hash tables. Given a key, the sketch allows one to reconstruct the value associated with it, with probabilistic bounds on the reconstruction accuracy. The achievable accuracy is a function of both the number of hash functions (H), and the size of hash tables (K). The baseline scheme uses a separate sketch data structure per distinct prefix length combination in all the dimensions.

Baseline variant 2: *Lossy Counting-based* solution (lc), which uses a deterministic, single-pass, sampling-based HH detection algorithm called *Lossy Counting* (see [27]). Lossy Counting uses two parameters: ϵ and ϕ, where $0 \leq \epsilon \ll \phi \leq 1$. At any instant, let N be the total number of items in the input data stream. *Lossy Counting* can correctly identify all heavy-hitter keys whose frequencies exceed ϕN. lc provides lower and upper bounds on the count associated with a heavy hitter. The gap between the two bounds is guaranteed to be at most ϵN. The space overhead for the algorithm is $O(\frac{1}{\epsilon} log(\epsilon N))$. The Lossy Counting algorithm can be modified to work with byte data instead of count data. All the complexity and accuracy results still apply except that we need to replace N with SUM. We use this adapted version in our evaluation.

We note that the algorithm in [12] is also based on Lossy Counting. So we expect its accuracy to be similar to that of lc. In addition, while their algorithm is normally much more efficient than lc, the worst-case amortized update cost is comparable to lc (the worst-case scenario can occur when the keys in the input stream are uniformly distributed, which can be caused by events like a distributed denial-of-service attack using spoofed source addresses). So although we do not directly compare against their algorithm, we expect the performance of lc to be indicative of the worst-case performance of their algorithm.

3.2 A trie-based solution to 1-d HHH detection

Our goal is to identify the prefixes (considering that we use the destination IP as the key) that are responsible for an amount of traffic that exceeds a given threshold. We would like to do so while maintaining minimal state and performing a minimum number of update operations for each arriving flow or packet.

The hierarchical nature of the problem reminds us of the classical IP lookup problem in which for every received packet the IP destination field in the packet header is used to search for a longest matching prefix in a set of given IP prefixes (also known as a routing table). The difference between our particular situation and the IP lookup problem is that in the IP lookup problem the set of prefixes is given as an input and is often *static*. In contrast, we need to generate *dynamically* (based on the packet arrival pattern) the set of prefixes that are associated with the heavy hitters.

Despite the difference, however, we are able to develop an effective solution to 1-d HHH detection by adapting an existing solution to the static IP lookup problem – the trie-based solution proposed by Srinivasan *et al.* [32].

Trie is a simple data structure. Each node in a one-bit trie has at most two child nodes, one associated with bit 0 and the other with bit 1. Srinivasan *et al.* [32] have extended on the basic idea of one-bit tries to create more refined multi-bit tries that are better suited for the IP lookup problem. Our algorithm is designed and implemented for m-bit tries, where each node of the trie has 2^m children, similar to the idea of the multi-bit tries. However for simplicity we describe our algorithm using one-bit tries.

The trie data structure. We maintain a standard trie data structure (as illustrated in Figure 1). Each node n in the trie is associated with a prefix $p*$ identified by the path between the root of the trie and the node. Array n.`child` contains pointers to the children of n. Field n.`depth` gives the depth of n. Field n.`fringe` indicates whether n is a fringe node – we consider n as a *fringe* node if after its creation, we see less than T_{split} amount of traffic associated with destination prefix p; otherwise, we consider n as an *internal* node. Field n.`volume` records the volume of traffic associated with prefix p that we see after n is created and before n becomes an internal node. Field n.`subtotal` gives the total volume of traffic for the entire subtrie rooted at n, excluding the portion already accounted for by n.`volume`. Fields n.`miss_copy` and n.`miss_split` represent estimated volume of traffic missed by node n (*i.e.*, traffic that is associated with prefix p but appears before the creation of n). The *copy-all* and the *splitting* rules are used to compute n.`miss_copy` and n.`miss_split`, respectively (details to follow). The last four volume related fields are used to estimate the total volume of traffic that is associated with prefix p. We will describe the estimation algorithm later in this section.

```
// vol_type is the data type for volume
typedef struct {
    trie *      child[·];     // child[i] points to the i-th child
    int         depth;        // the depth of this node
    boolean     fringe;       // true iff volume for entire subtrie < T_split
    vol_type    volume;       // volume of traffic trapped at this node
    vol_type    subtotal;     // total volume of traffic in all descendents
    vol_type    miss_copy;    // missed traffic (estimated by copy-all)
    vol_type    miss_split;   // missed traffic (estimated by splitting)
} trie;
```

Figure 1: The trie data structure

Updating the trie. Our data structure starts with a single node trie that is associated with the zero-length prefix $*$. The `volume` field associated with this node is incremented with the size of each arriving packet. When the value in this field exceeds T_{split}, we mark the node as internal and create one new child node associated with the prefix $0*$ or $1*$ that the incoming packet matches. The size of the current packet is then used to initialize the `volume` field in the newly created child node. The structure develops dynamically with the arrival of each new packet. This procedure is summarized in Figure 2.

```
int UPDATE_1D($key$, $value$)
    $n = root$
    while ($true$)
        if ($n$.fringe)
            if ($n$.volume $+ value < T_{\text{split}}$)
                $n$.volume $+= value$
                return $n$.depth $- 1$
            else
                $n$.fringe $= false$
                if ($n$.depth $= W$)
                    $n$.subtotal $= value$
                    return $n$.depth
                endif
            endif
        else if ($n$.depth $= W$)
            $n$.subtotal $+= value$
            return $n$.depth
        endif
        $index = get\_Nth\_bit(key, n$.depth $+ 1)$
        $c$ = get_child($n$,$index$)
        if ($c = NULL$)
            $c$ = create_child($n$, $index$)
        endif
        $n = c$
    endwhile
```

Figure 2: The update operation is very simple: walk down the trie until we reach a fringe node, and check if we can update its volume. if the updated volume is still below T_{split}, make the update and return; otherwise, mark the node as internal and continue walking down the trie. The actual implementation also includes some special handling when we reach the bottom of the trie (*i.e.*, we use up all bits in the key)

In Figure 3 we illustrate the update operation for a trie with T_{split} set to 10. The arriving packet has a Destination IP prefix of $100*$ and a size of 5 bytes. Figure 3 (a) shows the trie at the time of the packet arrival. The algorithm first performs a longest matching prefix operation on the trie and arrives at the node associated with prefix $10*$. Adding 5 bytes to the `volume` field of this node would make its value cross T_{split}. Therefore, the algorithm creates a new node associated with prefix $100*$ (*i.e.*, the child node associated with bit 0). The size of the current packet is used to initialize the `volume` field of the newly created node. Figure 3(b) shows the trie after the update operation.

One can see that our trie construction process guarantees that the values of the `volume` field in any internal node is always less than T_{split}. As a result, if we set $T_{\text{split}} = \epsilon SUM/W$, we can ensure that the maximum amount of traffic we miss as we dynamically drill down to the fringe is at most ϵSUM.

The time complexity of the operations described above is on the same order of magnitude as a regular IP lookup operation, *i.e.*, $O(W)$. For every packet arrival, we update at most one node in the trie. At most one new node is created during each update as long as the volume for the new item is below T_{split} (in case the volume exceeds T_{split}, we need to create an entire new branch all the way to the maximum depth W). It is easy to see that at each depth, there can be no more than $SUM/T_{\text{split}} = W/\epsilon$ internal nodes (otherwise the total sum over all the subtries rooted at those nodes would exceed SUM,

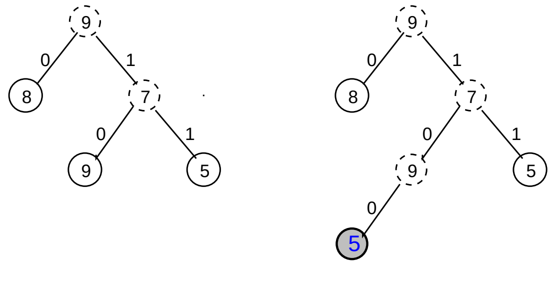


Figure 3: (a) shows the trie at the arrival of a packet of size 5 bytes and prefix $100*$. T_{split} is set to 10. (b) shows the trie after accounting for the packet. The newly created node is represented in grey. In both tries, dotted circles represent the internal nodes, while solid circles represent the fringe nodes.

```
vol_type COMPUTE_TOTAL_1D($n$)
    if ($n$.depth $\neq W$)
        $n$.subtotal $= 0$
        for (each child $c$ of $n$)
            if ( $c \neq NULL$ )
                $child\_total$ = COMPUTE_TOTAL_1D($c$)
                $n$.subtotal $+= child\_total$
            endif
        endfor
    endif
    return ($n$.volume $+$ $n$.subtotal)
```

Figure 4: The total volume associated with an internal node can be reconstructed recursively in a bottom-up fashion.

which is impossible). So the worst-case memory requirement of the data structure is $O(W^2/\epsilon)$.

Reconstructing volumes for internal nodes. In our trie building algorithm, every packet arrival results in at most one update. The update occurs at the node which is the most specific node representing the destination IP prefix (of the packet) at the time of the packet arrival. Therefore we need to reconstruct the volumes of the internal nodes at the end of the time interval. By delaying the reconstruction process to the end of the time interval, the reconstruction cost is amortized across the entire time interval.

To compute the volumes associated with all the internal nodes, we perform a recursive post-order traversal of the trie. In each recursive step the volume of the current node is computed as being the sum of the volume represented in the current trie node and its child nodes. This procedure is illustrated in Figure 4.

Estimating the missed traffic for each node. We note that because of utilizing T_{split} to guide the trie construction process the volumes represented in the internal nodes even after reconstruction are not entirely accurate. In order to more accurately estimate the volume associated with a given node, we also need to include an estimate of the missed traffic for that node. Below we consider three ways of estimating the missed traffic.

```
void COMPUTE_MISSED_1D($n$)
    for (each child $c$ of $n$)
        if ($c \neq NULL$ )
            $c$.miss_copy $=$ $n$.volume $+$ $n$.miss_copy
            $frac = (c$.subtotal $+$ $c$.volume$)/n$.subtotal
            $c$.miss_split $= (n$.volume $+$ $n$.miss_split$) \cdot frac$
            COMPUTE_MISSED_1D($c$)
        endif
    endfor
```

Figure 5: The missed traffic can be estimated in a top-down fashion (using either the *copy-all* or the *splitting* rule)

.

Copy-all: the missed traffic for a node N is estimated as the sum of the total traffic seen by the ancestors of node N in the path from node N to the root of the tree. Note that *copy-all* is conservative in that it copies the traffic trapped at a node to all its descendents. It always gives an upper bound for the missed traffic. Since our update operation maintains the invariant that every internal node n has n.volume below T_{split}, the estimate given by the *copy-all* rule is further upper bounded by *depth of the node* $\times T_{\text{split}}$.

No-copy: this is the other extreme that optimistically assumes the amount of missed traffic to be 0.

Splitting: the total contribution of missed traffic by a node n is split among all its children c in proportion to the total traffic for c. Essentially what this assumes is that the traffic pattern before and after the creation of a node are very similar, so we can predict missed traffic by proportionally splitting the traffic trapped at a node to all its children.

Both the *copy-all* and the *splitting* rule can be easily implemented by traversing the trie in a top-down fashion (as shown in Figure 5).

Detecting HHHs. Once we have an estimate of the missed traffic, we can combine it with the total amount of traffic we have seen and use the sum as input for HHH detection. The accuracy clearly depends on which rule we use: *copy-all* ensures that there is no false negative but there will be some false positives; *no-copy* ensures that there is no false positive but there may be some false negatives; *splitting* will have fewer false positives than *copy-all* and fewer false negatives than *no-copy*.

3.3 Detecting 2-d HHHs via Cross-Producting

We next consider the 2-dimensional HHH problem and develop a solution by adapting the cross-producting technique [33], which was originally proposed for solving the packet classification problem [33, 21, 3, 30, 20].

The high level idea of our solution is to execute our 1-dimensional algorithm described in Section 3.2, for each of the dimensions (IP destination, and IP source) and to use the length associated with the longest matching prefix nodes in each of the dimensions as an index into a data-structure that holds the volume data for the 2-dimensional HHHs.

In our solution, we maintain three data structures. Two tries are used to keep track of the 1-dimensional information, a $W \times W$ array H of hash tables is used to keep track of the 2-dimensional tuples. A tuple $< p_1, p_2 >$ comprises of the longest matching prefix in both the dimensions. The array is indexed by the lengths of the prefixes p_1 and p_2. In the case of IPv4 prefixes, for a 1-bit trie-based solution, $W = 32$.

Updating the summary data structure. For every incoming packet we first update the individual 1-dimensional tries, which return the longest matching prefix in each of the dimensions. This gives us two prefixes p_1 and p_2 with lengths l_1 and l_2 respectively. Next the two lengths are used as an index to identify the hash table $H[l_1][l_2]$. $< p_1, p_2 >$ is then used as a lookup key in the hash table $H[l_1][l_2]$. Subsequently, the volume field of the entry associated with the key is incremented. This process is repeated for every arriving packet. Figure 6 illustrates the basic algorithm.

For every packet three update operations are performed, one operation in each of the two 1-dimensional tries, and one operation in at most one of the hash-tables. This results in a very fast algorithm. The memory requirement in the worst case is $O((W^2/\epsilon)^2) = O(W^4/\epsilon^2)$, due to the use of cross-producting. But in practice, we expect the actual memory requirement to be much lower.

Reconstructing volumes for 2-d internal nodes. To compute the total volume for the internal nodes, we just need to add the volume for each element in the hash tables to all its ancestors. This can be implemented by scanning all the hash elements twice. During the first pass, for every entry e represented by key $< p_1, p_2 >$ (where p_1 and p_2 represent prefixes) with prefix lengths $< l_1, l_2 >$ we add

```
void UPDATE_CP($key_1, key_2, value$)
    $l_1$ = UPDATE_1D($trie_1, key_1, value$)
    $l_2$ = UPDATE_1D($trie_2, key_2, value$)
    **if** ($l_1 \geq 0 \wedge l_2 \geq 0$ )
        $p_1$ = prefix($key_1, l_1$)
        $p_2$ = prefix($key_2, l_2$)
        $H[l_1][l_2]$.update($< p_1, p_2 >, value$)
    **endif**
```

Figure 6: The update operation for Cross-Producting involves two 1-dimensional trie updates and one hash table update.

the volume associated with e to its left parent in the hash-map represented by key $< ancestor(p_1), p_2 >$ and lengths $< l_1 - 1, l_2 >$. Note that we start from entries with the largest l_1 and end with entries with the smallest l_1. Then in the second pass, we add the volume to right parent represented by the key $< p_1, ancestor(p_2) >$ and lengths $< l_1, l_2 - 1 >$. This time we start from entries with the largest l_2 and end with entries with the smallest l_2.

Estimating the missed traffic for each node. The algorithm is as follows. For each key (recall that the key is made up of the destination prefix and the source prefix) in the hash table traverse the individual tries to find the prefix represented by the key and return the missed traffic estimate obtained from the node (by applying either the *copy-all*, or the *splitting* rule as described in Section 3.2). The missed traffic is then estimated as the maximum of the two estimates returned by the two 1-d tries. Using the maximum preserves the conservativeness of *copy-all*.

3.4 Grid-of-Tries and Rectangle Search

The proposed scheme using the Cross-Producting technique is very efficient in time, however it can be potentially memory intensive in the worst case. We try to overcome this drawback by adapting two other well known algorithms for two-dimensional packet classification to our problem: Grid-of-Tries and Rectangle Search [33].

Just like Cross-Producting, both Grid-of-Tries and Rectangle Search have been applied in the packet classification context. This is not a coincidence. Conceptually, if we view each node as a rule, then finding nodes on the fringe becomes a packet classification problem.

However most packet classification algorithms are optimized for a relatively static rule set (through pre-computation), whereas in our context, we may need to dynamically maintain the fringe set. This may involve updating n nodes and possibly creating n new nodes. Despite the clear difference, we are able to adapt Grid-of-Tries and Rectangle Search to solve our problem. Since both algorithms have been well documented in the literature, we will only illustrate the basic idea and highlight the main difference. Interested readers should refer to [33, 31, 2] for further details on these algorithms.

3.4.1 Grid-of-Tries

The grid-of-tries data structure has been introduced by Srinivasan *et al.* [33] as a solution to the 2-dimensional packet classification problem. The data structure contains two levels of tries. The first level is associated with the IP destination prefixes in the classifier (a predefined rule set) while the second level tries are associated with IP source prefixes in the classifier.

For every valid prefix (P_1) node in the first level trie there is a pointer to a second level trie. The second level trie is created using all the prefixes (P_2) for which there is a rule P_1,P_2 in the classifier. For a complete description the reader is kindly directed to [33, 2]. As in the 1-dimensional HHH detection case, our grid-of-tries data structure is dynamically built based on the packet arrival pattern.

Constructing grid-of-tries for 2-d HHH detection. Each node in the data structure contains a pointer to each of its children. In addition each node in the first-level trie maintains a pointer to a second-level trie and each node in the second-level trie maintains a *jump pointer* (details to follow) for fast trie traversal. The thing to note is that there is only one first-level trie, but multiple second-level tries. Specifically, there is a second-level trie for each node in the first-level trie. Each node also stores a `volume` field associated with the volume of traffic that corresponds to all the packets having a prefix equal with the prefix of the node from the moment that the node is created till the moment when new child nodes are associated with the node.

Let us assume the existence of a current grid-of-tries structure at the given moment. New nodes and tries may be appended to the current grid-of-tries with the arrival of a new packet. First, a longest matching prefix (LMP) operation is executed in the first-level trie (using the destination prefix). A fringe node is always identified. Then same as in the case of our 1-d trie algorithm (described in section 3.2) if the volume associated with this node becomes greater than T_{split} then a new child node is created and associated with this node. As in the 1-d algorithm, the size of the current packet is used to initialize the `volume` field for the newly created child node. In addition to adding child nodes in the first-level trie, in our 2-d algorithm we must also initialize and associate a new second-level trie with each one of these newly created children. These second-level tries when first created are only initialized with a root node. The size of the current packet is used to increment the volume associated with the second-level trie that is associated with the new LMP in the first-level trie.

The arrival of a packet may also result in a situation where the node represented by the LMP in the second-level trie exceeds T_{split}. In this case a new child is created and associated with this node in the second-level trie in a way similar to the 1-dimensional HHH detection node creation process.

Every packet that arrives may contribute to multiple updates in the `volume` field of the nodes in the second dimension tries. To illustrate the update process let us consider the example in Figure 7, and the arrival of a packet with destination IP prefix $000*$, and source IP prefix $111*$ with a size of 4 bytes. T_{split} is set to 10 for this illustration. Figure 7 represents the grid-of-tries data structure at the time of the packet arrival. For the moment ignore the dotted lines in the figure. This arriving packet contributes to a modification in the value of the `volume` field in each one of the second-dimension tries associated with the LMP node in the first-dimension and all ancestors of this LMP node. Figure 8 shows the data structure after the update operation. The nodes that are affected by the update are shown in grey. To walk through the process, first a LMP operation was done in the first-level trie using the first prefix $000*$, and the value of the `volume` field associated with this LMP node is increment. We next follow the pointer to the second-level trie. Again we do a LMP operation in the second-level trie using the second prefix $111*$. Our search terminates with the node for prefix $1*$. If we were to add the size of the current packet to the volume associated with this node it would increase beyond T_{split}. We therefore create a new child node for this node. The size of the current packet is used to initialize the volume associated with the new child node for prefix $11*$ as this new node now represents the LMP. We must also update the second level tries associated with all the less specific prefixes of $000*$ namely $00*$, $0*$ and $*$.

In order to provide a fast update operation, each fringe node in the second-level trie contains a pre-computed *jump pointer*. Each fringe node in a second-level trie T_2 for prefix P_2 originating at prefix P_1 in the first-level trie maintains a jump pointer to the *same* prefix P_2 in a second-level trie that is associated with the direct ancestor of P_1. Note that the jump pointer discussed here can be maintained dynamically – whenever we create a node in the second-level trie associated with P_1, we also create a node for the second-level trie associated with the direct ancestor of P_1 (if not already present). In contrast, schemes discussed in the packet classification context are more complicated and require precomputation [2]. Utilizing jump pointers allows us to keep the time complexity within $O(W)$ as dur-

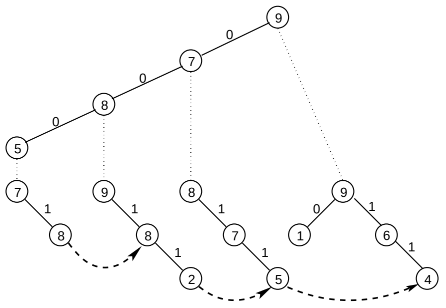


Figure 7: The grid-of-trie data structure at the time of a packet arrival. One can see a second-level trie is associated (connected by dotted lines in the figure) with each node in the first level trie. The dashed lines represent jump pointers (which are always between nodes with the same source prefix).

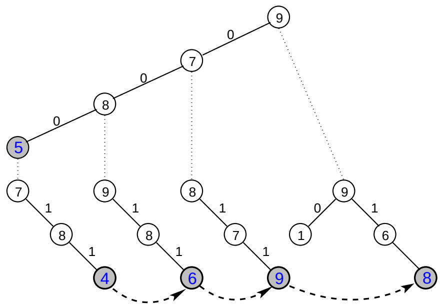


Figure 8: The grid-of-trie data structure after the update operation. The nodes to which we add the size of the current packet are shown in grey. The dashed lines represent jump pointers (which are always between nodes with the same source prefix).

ing the update process we can avoid having to restart the longest prefix matching problem at the root of every second-level trie (recall that we need to update every second-level trie associated with all ancestors of the longest matching prefix node in the path between the node and the root of the first-level trie). The dashed lines in Figure 7 and 8 represent jump pointers.

To ensure we only miss ϵSUM traffic in the worst case, we need to choose $T_{\text{split}} = \epsilon SUM/(2W)$. The space requirement is $O(W^2 \cdot (2W)/\epsilon) = O(2W^3/\epsilon)$.

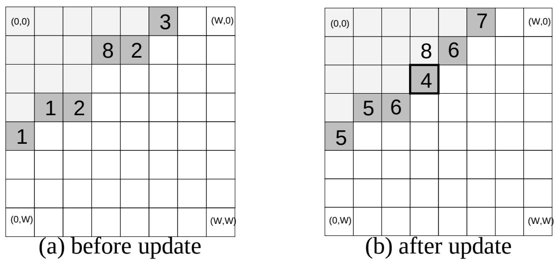
3.4.2 Rectangle Search

Rectangle Search [33] is another classic solution proposed for 2-dimensional packet classification. Like Grid-of-Tries, it can be adapted to solve the 2-dimensional HHH detection problem.

Conceptually, Rectangle Search does exactly the same thing as Grid-of-Tries – updating all the elements on the fringe and expanding it whenever necessary. The major difference lies in how the algorithm locates all the elements on the fringe. Grid-of-Tries does so using jump pointers. In the worst case, it requires $3W$ memory accesses, where W is the width of the key. Rectangle Search uses hash tables instead and requires $2W$ (hashed) memory accesses in the worst case.

The basic data structure for Rectangle Search is a set of hash tables arranged into a 2-dimensional array. More specifically, for each destination prefix length l_1 and source prefix length l_2, there is an associated hash table $H[l_1][l_2]$. Initially, only $H[0][0]$ contains an element $< *, * >$ with volume 0.

The update operation for a new tuple $< k_1, k_2 >$ (with value v) is illustrated in Figure 3.4.2. We first consider the case when v is below T_{split}, which is the common case as the total number



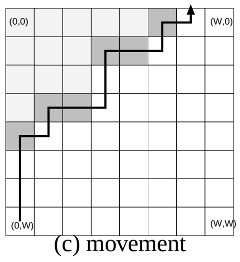


Figure 9: The update operation for rectangle search. The fringe nodes are in dark shade, and the internal nodes are in light shade. When a new tuple $< k1, k2 >$ (with value v) arrives, we start from the bottom left corner and move towards the upper right corner. T_{split} is set to 10. So a new element gets created.

of elements above the T_{split} is limited. The algorithm starts with $(l_1, l_2) = (0, W)$ (the lower left corner in Figure 3.4.2(c)). During each step, the algorithm checks if tuple $< p_1, p_2 >$ belongs to the hash table $H[l_1][l_2]$, where $p_i = prefix(k_i, l_i)$. If $< p_1, p_2 >$ does not exist in $H[l_1][l_2]$, we simply decrement l_2 by 1 (*i.e.*, move upwards in Figure 3.4.2(c)) and continue. Otherwise, we have found an element e. If e is a fringe node and $e.\texttt{volume} + v$ is below T_{split}, we simply add v to $e.\texttt{volume}$. Otherwise, either e is already an internal node (when updating some other descendents of e) or should become one after this update. In either case, we create a new element with key $< p_1, prefix(k_2, l_2 + 1) >$ and value v and insert it into $H[l_1][l_2+1]$. In case $l_2 = 0$ and e becomes a new internal node, then we also expand the fringe towards the right by creating an element with the key $< prefix(k_1, l_1+1), p_2 >$ and inserting it into $H[l_1 + 1][l_2]$. We then increment l_1 by 1 and continue (*i.e.*, move towards right in Figure 3.4.2(c)). The algorithm terminates whenever either $l_1 > W$ or $l_2 < 0$. Since during each step either we either increment l_1 by one or decrement l_2 by one, the algorithm takes at most $2W - 1$ steps to terminate.

When v is above T_{split}, the algorithm is virtually identical, except that for each l_1 we need to insert one element with value 0 into each hash table $H[l_1][j]$ ($l_2 < j < W$) and then one element with value v into hash table $H[l_1][W]$. In the worst case, this may create $(W + 1)^2$ new elements. But since the number of elements above T_{split} is small (below SUM/T_{split}), the amortized cost is quite low.

The pseudo code in Figure 10 illustrates the general idea for the update operation when v is below T_{split}. The actual implementation is more detailed due to issues like boundary cases.

Just like Grid-of-Tries, Rectangle Search requires $O(2W^3/\epsilon)$ space to guarantee an error bound of ϵSUM.

3.5 Lazy expansion

In all the algorithms described so far, whenever we receive an item $< k_1, k_2 >$ with value v above T_{split}, we will create state for all its ancestors $< p_1, p_2 >$ if they do not already exist. Such express expansion of the fringe has the advantage that it leads to less missed traffic for the fringe nodes and thus higher accuracy. However, it also requires a lot of space, especially when T_{split} is very small and there are a large number of items with value above it (this can happen, for instance, when the maximum depth of the trie is large). Here we

```
void UPDATE_RS($k_1$, $k_2$, $v$)
    $l_1 = 0$;  $l_2 = W$; // lower left corner
    while ( $l_1 \leq W \wedge l_2 \geq 0$ )
        $p_1 = prefix(k_1, l_1)$;  $p_2 = prefix(k_2, l_2)$
        $e = H[l1][l2].lookup(< p_1, p_2 >)$
        if ( undefined($e$) )
            $l_2--$ // moving upwards
        else
            if ($e$.fringe $\wedge$ $e$.volume $+ v < T_{\text{split}}$)
                // e remains a fringe node
                $e$.volume$+= v$
            else // e becomes internal
                insert an element into $H[l_1][l_2 + 1]$
                if ($e$.fringe $\wedge$ $l_2 = 0$)
                    insert an element into $H[l_1 + 1][l_2]$
                endif
                $e$.fringe $= false$
            endif
            $l_1{++}$ // moving towards right
        endif
    endwhile
```

Figure 10: The update operation for Rectangle Search.

introduce a simple technique, *lazy expansion* to significantly reduce the space requirement.

The basic idea for lazy expansion is very simple. Whenever we receive a large item with value v satisfying $v/T_{\text{split}} \in [k-1, k]$, we split it into k smaller items, each with value $v/k < T_{\text{split}}$. We then perform k separate updates. Since each item is below T_{split}, it will lead to the creation of no more than W elements. So long as $k < W$, we are guaranteed to reduce space requirement while still achieving the same deterministic worst-case accuracy guarantee. Meanwhile, we can modify the update operation to batch k updates together (by taking into account the multiplicities of the item). This avoids any increase in the update cost.

3.6 Compression

So far all our algorithms assume a fixed value for SUM. For many online applications, however, it may be desirable to set the threshold as a fraction of the actual total traffic volume for the current interval, which is not known until all the traffic is seen. Our strategy is to first use a small threshold based on some conservative estimate of the total traffic (*i.e.*, a lower bound), and increase the threshold when a large amount of additional traffic is seen. Note that as we increase the threshold, we need to remove all the nodes that should no longer exist under the new threshold. We refer to this as the *compression* operation.

The *compression* algorithm for the 1-d case is illustrated in Figures 11 and 12. We maintain a lower bound and an upper bound of the actual sum (SUM). Whenever the actual sum exceeds the upper bound, we perform the compression operation and then double the upper bound. The compression operation simply walks through the trie in a top down manner and removes the descendents of all the fringe nodes (according to the new threshold). The algorithms are more involved in 2-d case, but the high-level idea is very similar. We omit them for the interest of brevity. We make the following comments:

- In the worst case, compression can double the space requirement. It also adds some computational overhead. But the number of compression operations only grows logarithmically with the value of SUM. In practice, we can often get a reasonable prediction of the actual sum based on past history. So typically we just need a very small number of compressions.

- Compression can potentially provide a better accuracy bound. In particular, a node can potentially get created sooner than with a larger threshold, so the amount of missed traffic can be lower (but in the worst case, the accuracy guarantee still remains the same). We will demonstrate such effects later in Section 6.

- Compression also makes it possible to aggregate multiple data summaries (possibly for different data sources or created at different times or locations). For example, in the 1-d case, to merge two tries, we just need to insert every node in the second trie into the first trie, update the total sum and detection threshold, and then perform the compression operation (using the new detection threshold). Such aggregation capability can be very useful for applications like detecting distributed denial-of-service attacks.

```
initialization:
    $SUM^{\text{L}}$ = lower bound of actual $SUM$
    $SUM^{\text{U}} = 2 \cdot SUM^{\text{L}}$
    $T_{\text{split}} = \epsilon \cdot SUM^{\text{L}}/W$
upon each update:
    if ($SUM \geq SUM^{\text{U}}$)
        $SUM^{\text{L}} = SUM$
        $SUM^{\text{U}} = 2 \cdot SUM$
        $T_{\text{split}} = \epsilon * SUM^{\text{L}}/W$
        COMPUTE_TOTAL_1D($root$)
        COMPRESS_1D($root, T_{\text{split}}$)
    endif
```

Figure 11: We maintain a lower bound and an upper bound of the actual total traffic volume (SUM) and perform compression whenever the actual SUM exceeds the upper bound.

```
// assuming COMPUTE_TOTAL_1D has been called
void COMPRESS_1D($n$,$thresh$)
    if ( $n$.volume $+$ $n$.subtotal $< thresh$ )
        $n$.fringe $= true$
        $n$.volume $=$ $n$.volume $+$ $n$.subtotal
        $n$.subtotal $= 0$
        delete all descendents of $n$
    else
        for (each child $c$)
            COMPRESS_1D($c, thresh$)
        endfor
    endif
```

Figure 12: Compression is done in a top-down manner.

3.7 5-d HHH detection for network anomaly detection

We can use Rectangle Search and Grid-of-Tries as a building block to solve the general n-dimensional HHH detection problem and always result in a factor of W improvement over the brute-force approach. However, this may still be too slow for many applications.

Fortunately, for many practical applications, we do not need to deal with general HHH detection in all the fields. This is precisely the case for network anomaly detection, our primary motivating application. In this context, we need to handle 5 fields: (src_ip, dst_ip, src_port, dst_port, protocol). For protocol, we would typically require exact match (TCP, UDP, ICMP, others). For source or destination port, we can construct some very fat and shallow tree. For instance, we can use a 3-level tree, with level 0 being * (*i.e.*, don't care), level 1 being the application class (Web, chat, news, P2P, etc.), and level 2 being the actual port number. In addition, we typically only need to match on one of the port numbers (instead of their combination). Finally, we typically only care about port numbers for TCP and UDP protocols. Putting all these together, it often suffices to just consider the following 6 combinations in the context of network anomaly detection. For each combination, we have an array of grid-of-tries. So the update operation involves updating 6 tries.

src_port	*	TCP/UDP
src_app	*	TCP/UDP
*	dst_port	TCP/UDP
*	dst_app	TCP/UDP
*	*	protocol
*	*	*

4. APPLICATION TO SCALABLE CHANGE DETECTION

Change detection is a major component for statistical anomaly detection. The standard techniques for change detection include different smoothing techniques (such as exponential averaging), the Box-Jenkins ARIMA modeling [6, 7, 1] and wavelet-based [5, 4]. In the context of network applications, however, one often needs to deal with tens of millions of network time series and it is infeasible to apply standard techniques on per time series basis. To address the challenge, Krishnamurthy *et al.* [26] propose to perform scalable change detection on massive data streams through the use of sketch, a probabilistic data summary technique. Sketch-based change detection works very well when there is only a single fixed aggregation level. But if we want to apply it to find changes at all possible aggregation levels, we have to take a brute-force approach and run one instance of sketch-based change detection for every possible aggregation level, which can be prohibitive.

In this section, we demonstrate how we can perform scalable change detection for all possible aggregation levels by using our HHH detection algorithms as a pre-filtering mechanism. The basic idea is to extract all the HHH traffic clusters using a *small* HHH threshold ϕ in our HHH detection algorithms, reconstruct time series for each individual HHH traffic cluster, and then perform change detection for each reconstructed time series. Intuitively, if a cluster never has much traffic, then it is impossible to experience any significant (absolute) changes. So we expect our approach to capture most big changes so long as the HHH threshold ϕ is sufficiently small. We show later in Section 6.2 that this is indeed the case.

A major issue we need to address is how to deal with the reconstruction errors introduced by our summary data structure. The picture is further complicated by the increasing use of sampling in network measurements, which introduces sampling errors to the input stream. Lack of effective mechanisms to accommodate such errors can easily lead to false alarms (*i.e.*, detection of spurious changes). Our change detection method can accommodate both types of errors in a unified framework. It is quite general and can be applied to any linear forecast model, including various smoothing techniques and Box-Jenkins ARIMA modeling.

Below we present our method in the context of one specific change detection method: Holt-Winters, which has been successfully applied in the past for anomaly detection [8]. Given a time series $\{X_i\}$, the (non-seasonal) Holt-Winters forecast model maintains a separate smoothing baseline component S_i and a linear trend component T_i. There are two exponential smoothing parameters $\alpha \in [0, 1]$ and $\beta \in [0, 1]$.

$$S_i = \begin{cases} \alpha\, X_{i-1} + (1-\alpha)\,(S_{i-1} + T_{i-1}) & i > 2 \\ X_1 & i = 2 \end{cases} \quad (1)$$

$$T_i = \begin{cases} \beta\,(S_i - S_{i-1}) + (1-\beta)\, T_{i-1} & i > 2 \\ X_1 - X_0 & i = 2 \end{cases} \quad (2)$$

The forecast is simply $F_i = S_i + T_i$. The forecast error is then $E_i = X_i - F_i$. Big changes can be detected by looking for data points that significantly deviate from the forecast, *i.e.*, with forecast errors E_i exceeding the (time-varying) detection threshold DT_i. For online change detection, it is common to maintain an exponentially weighted moving average of $|E_i|$ and set DT_i to be some multiple of this smoothed deviation.

4.1 Extracting time series

Given a traffic cluster (with true traffic volume X_i in interval i), our summary data structure produces three different values by using different rules to calculate the amount of missed traffic: a lower bound X_i^{L} (using the *no-copy* rule), an upper bound X_i^{U} (using the *copy-all* rule), and an estimate X_i^{S} (using the *splitting* rule). Our experience with HHH detection suggests that X_i^{S} often gives the most accurate estimate (see Section 6.1). Therefore, we use time series $\{X_i^{\mathrm{S}}\}$ as the input for the Holt-Winters forecast model to obtain E_i^{S} and DT_i^{S}, which are estimates for the true forecast errors E_i and detection thresholds DT_i, respectively. We also use X_i^{L} and X_i^{U} to obtain tight bounds on the true forecast errors E_i as shown in Section 4.3.

4.2 Dealing with missing clusters

One important issue we need to deal with is the presence of missing clusters. A cluster may not appear in the summary structure for every interval. When this happens, we would still like to estimate its associated traffic volume, otherwise there will be a gap in the reconstructed time series. Fortunately, our summary structure allows us to conveniently obtain such estimates. For example, given a 2-d missing cluster with key $< p_1, p_2 >$, conceptually all we need to do is to insert a new element with key $< p_1, p_2 >$ and value 0 into the summary data structure, which will result in one or more newly created fringe nodes. We can then obtain estimates for the first newly created fringe node and use them as the corresponding estimates for $< p_1, p_2 >$. After this, we can then remove all the newly created nodes through *compression*. Note that in the final implementation, we do not need to actually create the new fringe nodes and then remove them – we just need to do a lookup to find the first insertion position.

4.3 Obtaining bounds on forecast errors

Let the use of superscript $^{\mathrm{L}}$ and $^{\mathrm{U}}$ on a variable denote the lower and upper bounds for the variable, respectively. For example, X_i^{L} denotes the lower bound for X_i. Below we show how to compute E_i^{L} and E_i^{U}, the bounds for the true forecast errors E_i.

A naive solution. At the first glance, it seems rather straightforward to compute E_i^{L} and E_i^{U}—we can directly apply (1) and (2) to recursively compute bounds for S_i, T_i and then use them to form bounds for F_i and E_i. More specifically, we have

$$S_i^{\mathrm{U}} = \alpha\, X_{i-1}^{\mathrm{U}} + (1-\alpha)\,(S_{i-1}^{\mathrm{U}} + T_{i-1}^{\mathrm{U}}) \quad (3)$$

$$S_i^{\mathrm{L}} = \alpha\, X_{i-1}^{\mathrm{L}} + (1-\alpha)\,(S_{i-1}^{\mathrm{L}} + T_{i-1}^{\mathrm{L}}) \quad (4)$$

$$T_i^{\mathrm{U}} = \beta\,(S_i^{\mathrm{U}} - S_{i-1}^{\mathrm{L}}) + (1-\beta)\, T_{i-1}^{\mathrm{U}} \quad (5)$$

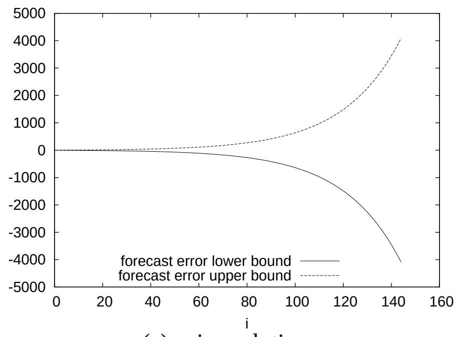
$$T_i^{\mathrm{L}} = \beta\,(S_i^{\mathrm{L}} - S_{i-1}^{\mathrm{U}}) + (1-\beta)\, T_{i-1}^{\mathrm{L}} \quad (6)$$

$$F_i^{\mathrm{U}} = S_i^{\mathrm{L}} + T_i^{\mathrm{L}} \qquad F_i^{\mathrm{L}} = S_i^{\mathrm{U}} + T_i^{\mathrm{U}} \quad (7)$$

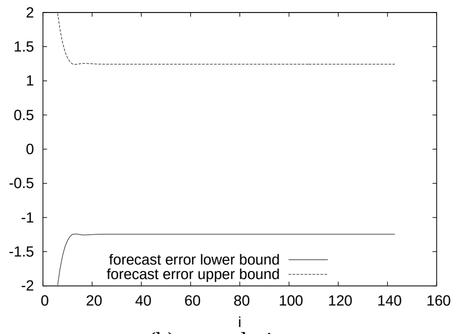
$$E_i^{\mathrm{U}} = X_i^{\mathrm{U}} - F_i^{\mathrm{L}} \qquad E_i^{\mathrm{L}} = X_i^{\mathrm{L}} - F_i^{\mathrm{U}} \quad (8)$$

Unfortunately, reconstruction errors can accumulate exponentially with this approach and cause the resulted bounds E_i^{L} and E_i^{U} to be too loose to be useful. This is evident in Figure 13(a), which shows the forecast error bounds produced by the naive solution when $X_i^{\mathrm{L}} = 0$ and $X_i^{\mathrm{U}} = 1$.

Our solution. We can obtain tight bounds by directly representing S_i and T_i as linear combinations of X_j ($j \le i$) and then incorporating the bounds X_i^{L} and X_i^{U}. More specifically, let $S_i = \sum_{j=1}^{i-1} s[i,j] X_i$ and $T_i = \sum_{j=1}^{i-1} t[i,j] X_j$. From (1) and (2), we can



(a) naive solution



(b) our solution

Figure 13: Forecast error bounds when $X_i^{\mathbf{L}} = 0, X_i^{\mathbf{U}} = 1$ ($\alpha = 0.5, \beta = 0.25$)

compute $s[i,j]$ and $t[i,j]$ recursively as follows:

$$s[i,j] = \begin{cases} \alpha & j = i-1 \\ (1-\alpha)\,(s[i-1,j] + t[i-1,j]) & j < i-1 \end{cases}$$
$$t[i,j] = \beta\,(s[i,j] - s[i-1,j]) + (1-\beta)\,t[i-1,j]$$

We can prove by induction that $s[i,j] = s[i-1,j-1]$ and $t[i,j] = t[i-1,j-1]$ for $\forall j > 2$ (proof omitted for the interest of brevity). So when we increment i, we only need to compute $s[i,j]$ and $t[i,j]$ for $j \leq 2$. Once we have $s[i,j]$ and $t[i,j]$, let $f[i,j] = s[i,j] + t[i,j]$. We then compute the forecast error bounds $E_i^{\mathbf{L}}$ and $E_i^{\mathbf{U}}$ as

$$E_i^{\mathbf{U}} = X_i^{\mathbf{U}} - \sum_{j:\ f[i,j]>0} f[i,j]\cdot X_j^{\mathbf{L}} - \sum_{j:\ f[i,j]<0} f[i,j]\cdot X_j^{\mathbf{U}}$$
$$E_i^{\mathbf{L}} = X_i^{\mathbf{L}} - \sum_{j:\ f[i,j]>0} f[i,j]\cdot X_j^{\mathbf{U}} - \sum_{j:\ f[i,j]<0} f[i,j]\cdot X_j^{\mathbf{L}}$$

As shown in Figure 13(b), our solution yields very tight bounds.

Note that the above solution requires keeping the entire interval series $[X_i^{\mathbf{L}}, X_i^{\mathbf{U}}]$. Our solution is simply to ignore the remote past. This is reasonable as the use of exponential smoothing means the remote past has very little effect on predicting the future. That is, $f[i,j]$ becomes very small when $i-j$ is sufficiently large. As a result, we only need to keep state for the most recent few intervals for each flow.

4.4 Testing for signifi cant changes

Recall that in Section 4.1 we apply time series analysis on $X_i^{\mathbf{S}}$ to compute $E_i^{\mathbf{S}}$ and $DT_i^{\mathbf{S}}$; in Section 4.3 we show how to compute the forecast error bounds. To accommodate the reconstruction errors introduced by the summary data structure, our detection criteria combines both $DT_i^{\mathbf{S}}$ and the forecast error bounds $E_i^{\mathbf{L}}$, $E_i^{\mathbf{U}}$. More specifically, we report a significant change whenever the two intervals $[E_i^{\mathbf{L}}, E_i^{\mathbf{U}}]$ and $[-DT_i^{\mathbf{S}}, DT_i^{\mathbf{S}}]$ do not overlap, *i.e.*, $[E_i^{\mathbf{L}}, E_i^{\mathbf{U}}] \cap [-DT_i^{\mathbf{S}}, DT_i^{\mathbf{S}}] = \emptyset$.

4.5 Dealing with sampling errors

Network measurements are increasingly subject to sampling. This introduces inherent variability into the traffic metrics under study. This section describes how the effects of sampling variability can be accommodated within our framework.

The idea is to represent each sampled measurement in the form `(key, value, var)` where `value` is an unbiased usage estimate (*e.g.* of bytes or packets in a flow) arising from sampling, and `var` is a sampling variance associated with the estimate. In this framework, the values to be estimated are considered as fixed rather than statistical quantities. Conditioned upon these values, the sampling decisions can be assumed independent. Hence when measurements are aggregated, the variance of the aggregate is taken to be the sum of the individual variances.

The aggregate variance can then be used to attach error bars to time series of a heavy hitters aggregate. We just need to maintain an estimate of the variance. This is easy because the variance can be updated in exactly the same way as the value:

```
whenever n.value = n.value + value
we do    n.var   = n.var + var
```

In the end, besides obtaining $X^{\mathbf{L}}$, $X^{\mathbf{U}}$, $X^{\mathbf{S}}$ for each cluster, we also have the corresponding estimates for aggregated sampling variance: $V^{\mathbf{L}}$, $V^{\mathbf{U}}$, $V^{\mathbf{S}}$.

We can then replace $X^{\mathbf{L}}$ and $X^{\mathbf{U}}$ with $X_*^{\mathbf{L}} - s\left(V^{\mathbf{U}}\right)^{0.5}$ and $X_*^{\mathbf{U}} = X^{\mathbf{U}} + s\left(V^{\mathbf{U}}\right)^{0.5}$, respectively. We can make s sufficiently large so that the probability for any actual value to fall outside the interval $[X_*^{\mathbf{L}}, X_*^{\mathbf{U}}]$ is extremely low (for example, using the 6 sigma rule if we the sampling error is close to Gaussian). We can then use $X_*^{\mathbf{L}}$ and $X_*^{\mathbf{U}}$ together with $X^{\mathbf{S}}$ in our earlier analysis.

Due to space limit, we do not explicitly show how the estimate `value` and its variance `var` are calculated when working with sampled flow statistics. Details can be found in [16, 15].

5. EVALUATION METHODOLOGY

We evaluate our HHH detection algorithms along a number of dimensions to measure their accuracy and resource (space and time) requirements. We use the following accuracy metrics.

- *False Positive* (FP) measures the number of entities that the algorithm incorrectly identifies as Hierarchical Heavy Hitters.

- *False Negative* (FN) measures the number of Hierarchical Heavy Hitters entities that the algorithm fails to identify as such.

- *Error estimate* (ES) for a HHH cluster is measured as the difference between the actual volume and the volume estimated by the algorithm.

For the accuracy experiments, we compute the FP, FN and ES values as follows: an offline evaluation computes the exact volumes for every multidimensional cluster, and given a value of ϕ, determines the true set of HHH clusters and their actual volumes. Examining the differences in set membership with the HHH set output by the online HHH detection algorithms yields the FP and FN. For each correctly identified HHH cluster, the difference between the actual and estimated volume yields ES.

We use the following resource metrics:

- *Space Overhead* : measured in terms of the number of entries in the two types of data structures involved: (a) array and (b) hash table.
- *Computation Overhead* : measures the runtime overheads of different HHH detection algorithms.in terms of the following three types of operations: (a) *lookup/update, i.e.*, get an entry from an array (via array indexing) or a hash table (via a hash table lookup) and possibly update its value; (b) *insertion, i.e.*, inserting new entries into an array or a hash table; (c) *deletion, i.e.*, deleting entries from an array or a hash table.

We use the following naming conventions for the different 2-d HHH detection algorithms: Cross Producting (cp), Grid-of-Tries (got), Rectangle Search (rs). cp^*, got^*, and rs^* are the corresponding variants with the *lazy expansion* optimization enabled. We compare these techniques against the three baseline HHH detection algorithms described in Section 3.1. For the sketch-based technique sk, recall that the accuracy bound depends on both H and K. In the evaluations, we set $K = 10/\epsilon$ and

$$H = \log(SUM \cdot (1 + 32/gran)^2/\delta)/\log(K\epsilon),$$

where $gran$ is the granularity we are using (*e.g.*, $gran = 8$ indicates we only consider prefix lengths $0, 8, 16, 24$ and 32), and SUM is the total traffic volume. This ensures that with probability $1 - \delta$ the method gives no false positives (the analysis is similar to the proof of Theorem 6 in [14]). In our evaluation we set $\delta = 0.01$. We also tested a less expensive solution $sk2$ that uses $H = 2$.

5.1 Dataset description

We use multiple large netflow traces collected from a tier-1 ISP to drive the evaluations of our algorithms (see Table 1).

trace	duration	#routers	# records	volume
ISP-100K	3 min	1	0.10 M	66.48 MB
ISP-1day	1 day	2	332.26 M	223.51 GB
ISP-1mon	1 month	2	7457.07 M	5.17 TB

Table 1: Data Description: Network Traces used

6. RESULTS

6.1 Evaluation of HHH detection

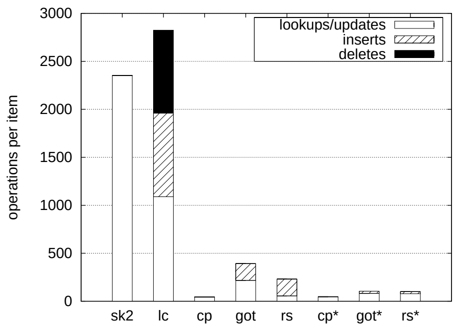
6.1.1 Resource Effi ciency

We first compare the amortized runtime costs of different HHH detection algorithms. Figure 14 compares the average number of operations for each newly arrived item using different algorithms and granularities on trace ISP-100k. Clearly, all our algorithms significantly outperform the brute-force solutions by orders of magnitude. In addition, for high resolution (i.e., $gran = 1$), the use of lazy expansion further reduces the runtime costs significantly for both got and rs. This is not surprising, as lazy expansion can significantly reduce the number of nodes to be created, resulting in a much smaller summary structure and thus runtime costs.

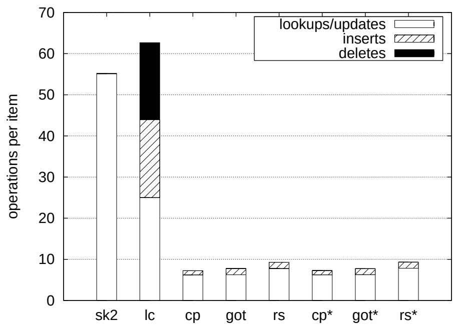
We next evaluate the space requirements. To better illustrate the behavior of these algorithms under different granularities, we normalize the space cost by $1/\epsilon \cdot (32/gran)^2$, the maximum possible number of flows whose traffic volume exceeds ϵSUM.

The results are summarized in Figures 15(a)-(b). Across both granularities, $sk2$ has the highest space requirement. Among the proposed algorithms, the pair got and rs have very similar space requirements, as do their counterpart pair got^* and rs^* that use lazy expansion.

For high resolution ($gran = 1$, see Figure 15(a)), got and rs have substantially higher space requirements than the existing lc algorithm. However, the use of lazy expansion in got^* and rs^* results



(a) trace = ISP-100k, gran = 1, $\epsilon = 0.001$



(b) trace = ISP-100k, gran = 8, $\epsilon = 0.001$

Figure 14: Amortized runtime costs for different algorithms. The total sum is assumed to be given in advance. The cost for sk (not shown) is 5.5 times more than sk2 due to the use of 11 instead of 2 tables per sketch.

in substantially smaller space requirements that are comparable to that for lc. For example, the space requirement for rs^* is just 14% of that for rs. Cross-Producting has the least overhead, both with and without lazy expansion.

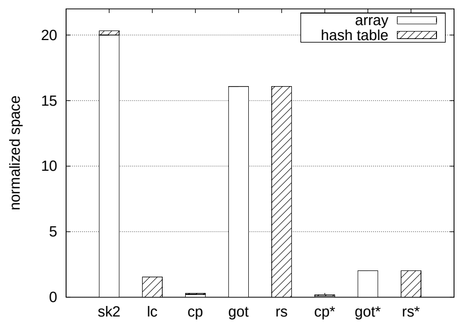
For the low resolution scenario ($gran = 8$, see Figure 15(b)), lazy expansion does not have any noticeable impact on the space usage of the proposed algorithms, and lc has the least space requirement. Note that the values in Figures 15(a)-(b) represent only conservative estimates of the space usage, as they depict only the space required by the hash table or array entries required in each approach, and ignore any auxiliary overhead associated with maintaining those data structures. For instance, rs, rs^* and lc use hash tables and thus require additional space to maintain the keys. An array-based approach like got or got^* does not have this additional overhead. Hence the actual difference between the space requirement of got^* and lc is smaller than shown in Figure 15(b). A more accurate space comparison should also account for these extra overheads.

The above plots demonstrate that lazy expansion results in both low space usage and low computation overhead. In the remainder of the evaluations, we shall use the lazy expansion variants (got^*, rs^*, cp^*) of our proposed algorithms.

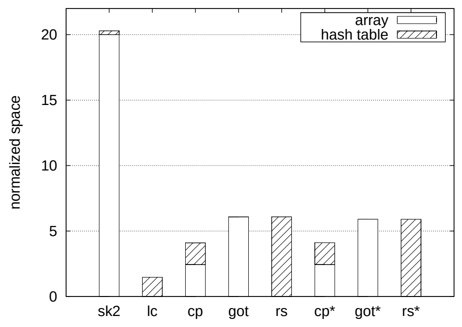
6.1.2 Accuracy

A number of factors determine the accuracy of the proposed algorithms – we consider each of them in turn. Note that under the same situation, got and rs always provide identical estimates of the count associated with a cluster, and hence have identical accuracy (FN, FP and ES) measures. In the following accuracy evaluations, we therefore only present results for got^*, with the knowledge that the accuracy-related conclusions for got^* apply identically to rs^*.

First we consider the impact of the three heuristics *copy-all, no-copy,* and *splitting* (introduced in Section 3.2) for estimating the overall traffic corresponding to a cluster. Figures 16(a)-(b) com-

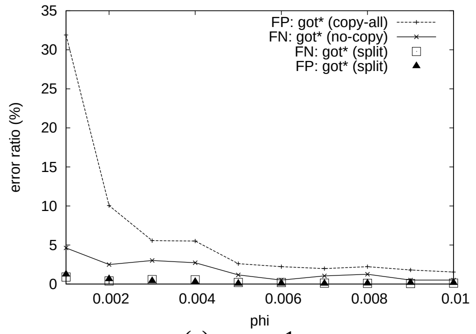


(a) trace = ISP-100k, gran = 1, $\epsilon = 0.001$

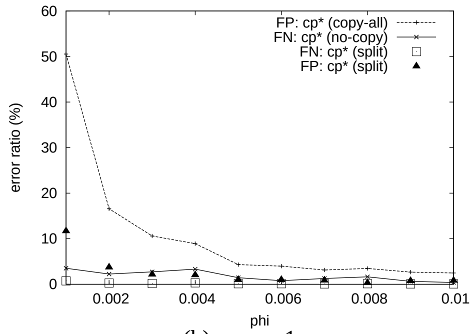


(b) trace = ISP-100k, gran = 8, $\epsilon = 0.001$

Figure 15: Normalized space costs of different algorithms. (normalized space cost = total space cost / $[1/\epsilon \cdot (32/gran)^2]$). The cost for sk (not shown) is 5.5 times more than that for sk2.



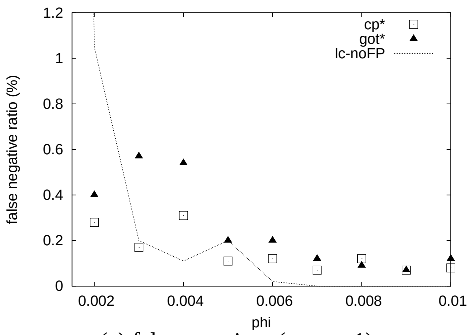
(a) gran = 1



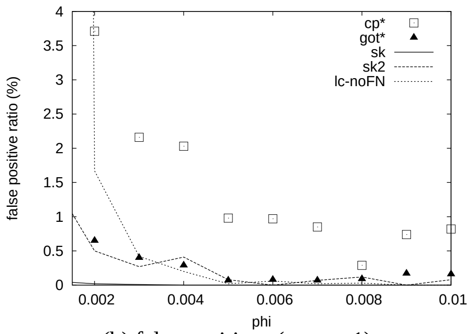
(b) gran = 1

Figure 16: Comparison between 3 detection criteria

pare the accuracy for the three heuristics for the 2-d HHH detection techniques got^* and cp^*, respectively, as a function of the HHH threshold ϕ, for $gran = 1$. Recall that by definition, *copy-all* has $FN = 0$, and *no-copy* has $FP = 0$ (the corresponding plots are omitted from the figures for better readability). The plots show that across all combinations of HHH detection algorithm and missing traffic estimation heuristic, both the FP and FN values are higher for smaller ϕ and and decrease for larger ϕ. FP for *copy-all* is substantially higher than for the other 2 heuristics, for both got^* and cp^*, particularly for small ϕ. Note also that *no-copy* has the worst FN among the three copy schemes. Given the low FP and FN for *splitting*, in the remainder of the evaluations, we focus on this heuristic.



(a) false negatives (gran = 1)

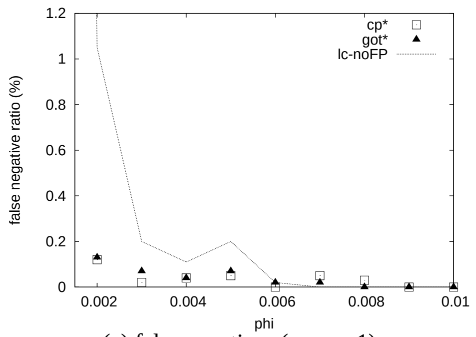


(b) false positives (gran = 1)

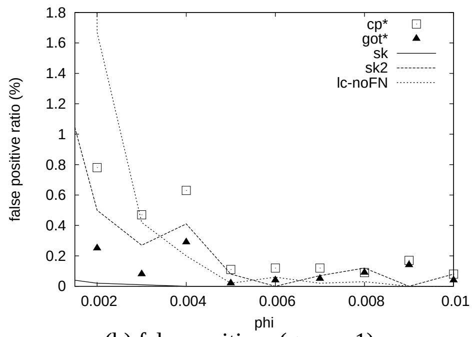
Figure 17: Accuracy without compression

We next examine the impact of the *compression* technique (see Section 3.6) on the accuracy. Figures 17(a)-(b) respectively plot the FN and FP as a function of ϕ, for different HHH detection algorithms, when *compression* is not used. Figures 18(a)-(b) present the corresponding plots when *compression* is used. Note that the *Lossy Count* based baseline HHH detection algorithm lc can be configured to detect HHHs using either a lower bound estimate (which ensures $FP = 0$), or an upper bound estimate (which ensures $FN = 0$) of the actual volume of each HHH; we use lc-noFP and lc-noFN to refer to these two configurations of lc, respectively. The set of plots reveal that *compression* significantly improves both and FN, for both cp^* and got^*. The cause of this behavior can be traced to the way *compression* works. Recall that the *compression* technique begins with a small initial estimate of the total volume – its expansion threshold is therefore smaller initially. Hence cp^* (also got^*) with *compression* may create a node for a HHH cluster and begin accounting for its traffic at an earlier instant, and therefore miss less counts for the cluster. This contributes to the increased accuracy. A second point to take away from the graphs is that the FP and FN values are low and comparable for the baselines and the proposed schemes, even without *compression*.

The detailed evaluations above are all based on the ISP-100K trace. We next use the much larger one month long trace ISP-1mon to measure the accuracy of got^* across the one-month period. We present the cummulative distribution of the FP and FN for two different routers in Figures 19(a)-(b). The plots show that the bulk of the FP and FN values are very low, for both routers.



(a) false negatives (gran = 1)



(b) false positives (gran = 1)

Figure 18: Accuracy with compression.

Algorithm	Normalized Error Estimates (%)			
	Max.	99 percentile	90 percentile	Med.
sk	4.30	0.64	0.06	0.00
$sk2$	239.83	101.27	2.15	0.00
lc-noFN	75.32	41.48	16.25	2.01
lc-noFP	97.67	71.70	30.12	0.28
got^*-split	7.26	3.28	1.52	0.40

Table 2: Normalized Error Estimates (absolute value) for the one month trace ISP-1mon (gran = 1).

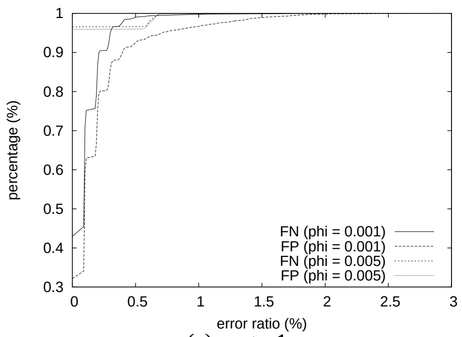
The FP and FN metrics measure an algorithm's ability to correctly identify HHH clusters. We are also interested in the accuracy of the the estimated volumes for the HHH clusters. Table 2 shows the empirical distribution of the absolute value of ES (defined in Section 5) normalized by $1/\epsilon$, for $gran = 1$. The values indicate that got with lazy expansion and compression has significantly lower ES than lc-noFN, lc-noFP and $sk2$. Only the baseline algorithm sk seems to have slightly better ES values that got^*. However, the computation cost for sk is significantly higher (recall Figure 14).

In summary, the key conclusions from the HHH evaluations are: (i) The techniques *lazy expansion*, *splitting* and *compression* are effective and should be used. (ii) compared to the baseline algorithms, the proposed algorithms $got*$, $cp*$, and $rs*$ have orders of magnitude smaller run-time costs, comparable or smaller space requirements, and comparable FN and FP values. Also the algorithm got had substantially lower volume reconstruction error values than the baselines – the only exception being sk, for which got had slightly worse ES.

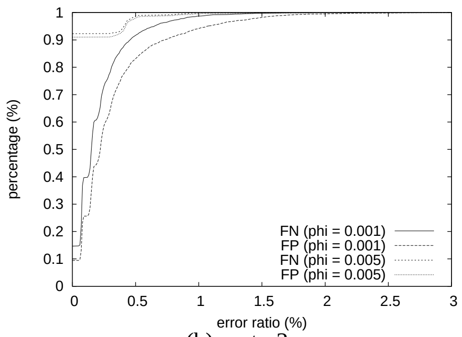
6.2 Evaluation on change detection

Figure 20 summarizes the overlap percentage between top N biggest changes reported by online and offline algorithms. The overlap ratio is always above 97% even for very large N. For N below 100, the top N lists produced by the two algorithms often differ by no more than one element.

Figure 21 illustrates the effects of smart sampling [16, 15] on accuracy. With a sampling threshold of 300KB, we are able to reduce the number of flow records to be processed by a factor of 12. Yet the accuracy still consistently remains above 90%.



(a) router1



(b) router2

Figure 19: Cummulative distribution of error ratios over an entire month (algo = $got*$, $\epsilon = 0.001$).

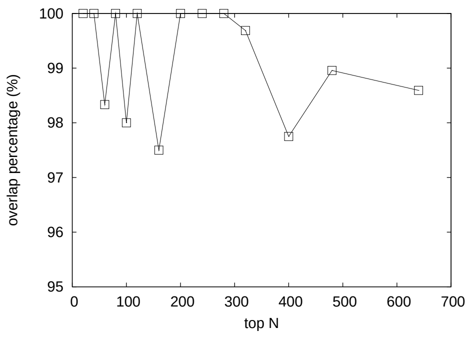


Figure 20: Overlap between top N biggest changes reported by online and offline algorithms for router2 in trace ISP-1day. Online algorithms uses got* with compression and lazy expansion ($\epsilon = 0.001$, $\phi = 0.001$).

7. CONCLUSIONS

In this paper, we present several efficient streaming algorithms for detecting multidimensional hierarchical heavy hitters. These algorithms are based on adaptive synopsis data structures that hierarchically organize the traffic into its most active components. The algorithms are much more efficient than existing algorithms, and provide data-independent deterministic accuracy guarantees on traffic estimates for the multidimensional hierarchical heavy hitters. Our motivating application is network anomaly detection, and we use robust techniques to detect changes among such heavy hitters. Our

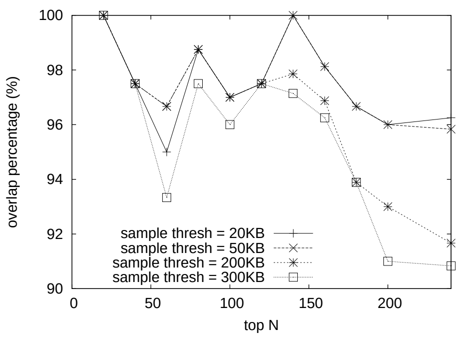


Figure 21: Overlap between top N biggest changes reported by online and offline algorithms for router2 in trace ISP-1day. Smart sampling with threshold between 20KB and 300KB is used (threshold = 300KB reduces the number of records to be processed by a factor of 12).

techniques can accommodate variability due to sampling that is increasingly used in network measurement. Evaluation using real Internet traces collected at a Tier-1 ISP suggests that these techniques are remarkably accurate and efficient. Our results are promising and point to the potential of using our algorithms as a building block for network anomaly detection and traffic measurement in large networks. We are developing a prototype anomaly detection system that embodies the algorithms developed in this paper.

Acknowledgments

We are grateful to Bill Aiello, Chuck Kalmanek, Muthu Muthukrishnan, Divesh Srivastava, and Mikkel Thorup for thought-provoking discussions. We would also like to thank Florin Baboescu, Flip Korn, George Varghese, and the anonymous reviewers for giving us valuable comments.

8. REFERENCES

[1] H. Arsham. Time series analysis and forecasting techniques. http://obelia.jde.aca.mmu.ac.uk/resdesgn/arsham/opre330Forecast.htm.

[2] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to CAMs. In *INFOCOM*, 2003. http://citeseer.ist.psu.edu/baboescu03packet.html.

[3] F. Baboescu and G. Varghese. Scalable packet classification. In *Proc. ACM SIGCOMM*, 2001. http://citeseer.ist.psu.edu/baboescu01packet.html.

[4] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, Marseille, France, November 2002.

[5] P. Barford and D. Plonka. Characteristics of network traffic fbw anomalies. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, San Francisco, CA, November 2001.

[6] G. E. P. Box and G. M. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, 1976.

[7] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis, Forecasting and Control*. Prentice-Hall, Englewood Cliffs, 1994.

[8] J. Brutlag. Aberrant behavior detection in time series for network monitoring. In *Proc. of the 14th USENIX System Administration Conference (LISA XIV)*, New Orleans, LA, December 2000.

[9] C. Chen and L.-M. Liu. Forecasting time series with outliers. *Journal of Forecasting*, 12:13–35, 1993.

[10] Cisco. *Random Sampled NetFlow*. http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/123newft/123t/123t_2/nfstatsa.pdf.

[11] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *International Conference on Very Large Data Bases*, 2003.

[12] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *Proc. ACM SIGMOD*, June 2004.

[13] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. In *Proc. ACM PODC '2003*, July 2003.

[14] G. Cormode and S. Muthukrishnan. Improved data stream summaries: The count-min sketch and its applications. In *Journal of Algorithms*, 2004. In press. http://dimacs.rutgers.edu/∼graham/pubs/cm-full.pdf.

[15] N. Duffield and C. Lund. Predicting resource usage and estimation accuracy in an IP fbw measurement collection infrastructure. In *ACM SIGCOMM Internet Measurement Workshop*, Miami Beach, FL, October 2003.

[16] N. Duffield, C. Lund, and M. Thorup. Charging from sampled network usage. In *ACM SIGCOMM Internet Measurement Workshop*, San Francisco, CA, November 2001.

[17] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, August 2002.

[18] C. Estan and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[19] F. Feather, D. Siewiorek, and R. Maxion. Fault detection in an ethernet network using anomaly signature matching. In *Proc. ACM SIGCOMM*, 1993.

[20] A. Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *INFOCOM (3)*, pages 1193–1202, 2000. http://citeseer.ist.psu.edu/feldmann00tradeoffs.html.

[21] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proc. ACM SIGCOMM*, pages 147–160, 1999. http://citeseer.ist.psu.edu/gupta99packet.html.

[22] C. Hood and C. Ji. Proactive network fault detection. In *Proc. IEEE INFOCOM '97*, Kobe, Japan, April 1997.

[23] K. J. Houle, G. M. Weaver, N. Long, and R. Thomas. Trends in Denial of Service Attack Technology. http://www.cert.org/archive/pdf/DoS_trends.pdf.

[24] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *Proceedings of the World Wide Web Conference*, Honolulu, Hawaii, May 2002. http://www.research.att.com/∼bala/papers/www02-fc.html.

[25] I. Katzela and M. Schwartz. Schemes for fault identification in communication networks. *IEEE/ACM Transactions on Networking*, 3(6):753–764, December 1995.

[26] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proc. ACM/USENIX Internet Measurement Conference*, 2003. http://www.research.att.com/∼yzhang/papers/nad-imc03.pdf.

[27] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *International Conference on Very Large Data Bases*, 2002.

[28] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm. Technical report, CAIDA, February 2003. http://www.cs.berkeley.edu/∼nweaver/sapphire/.

[29] S. Muthukrishnan. Data streams: Algorithms and applications, 2003. Manuscript based on invited talk from *14th SODA*.

[30] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proc. ACM SIGCOMM*, 2003. http://citeseer.ist.psu.edu/singh03packet.html.

[31] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proc. ACM SIGCOMM*, pages 135–146, 1999. http://citeseer.ist.psu.edu/srinivasan99packet.html.

[32] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. In *ACM Transactions on Computer Systems*, 1999.

[33] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proc. ACM SIGCOMM*, 1998. http://citeseer.ist.psu.edu/srinivasan98fast.html.

[34] R. S. Tsay. Outliers, level shifts, and variance changes in time series. *Journal of Forecasting*, 7:1–20, 1988.

[35] A. Ward, P. Glynn, and K. Richardson. Internet service performance failure detection. *Performance Evaluation Review*, August 1998.