FaRNet: Fast Recognition of High-Dimensional Patterns from Big Network Traffic Data

Ignasi Paredes-Oliva[a,*], Pere Barlet-Ros[a], Xenofontas Dimitropoulos[b]

[a] *UPC BarcelonaTech*
Jordi Girona 1-3, 08034 Barcelona, Spain
[b] *ETH Zurich*
Gloriastrasse 35, 8092 Zurich, Switzerland

Abstract

Extracting knowledge from big network traffic data is a matter of foremost importance for multiple purposes including trend analysis, network troubleshooting, capacity planning, network forensics, and traffic classification. An extremely useful approach to profile traffic is to extract and display to a network administrator the multi-dimensional hierarchical heavy hitters (HHHs) of a dataset. However, existing schemes for computing HHHs have several limitations: 1) they require significant computational resources; 2) they do not scale to high dimensional data; and 3) they are not easily extensible. In this paper, we introduce a fundamentally new approach for extracting HHHs based on generalized frequent item-set mining (FIM), which allows to process traffic data much more efficiently and scales to much higher dimensional data than present schemes. Based on generalized FIM, we build and thoroughly evaluate a traffic profiling system we call *FaRNet*. Our comparison with AutoFocus, which is the most related tool of similar nature, shows that *FaRNet* is up to three orders of magnitude faster. Finally, we describe experiences on how generalized FIM is useful in practice after using *FaRNet* operationally for several months in the NOC of GÉANT, the European backbone network.

Keywords: Network Operation and Management, Traffic Profiling, Data Mining

1. Introduction

In recent years, the Internet traffic mix has changed dramatically. Mobile applications, social networking, peer-to-peer applications and streaming services are only a few examples of the ever-growing list of applications that mold Internet traffic today. Furthermore, existing applications continuously change their

*Corresponding author (Tel: +34 934017182, Fax: +34 934017055)
Email addresses: iparedes@ac.upc.edu (Ignasi Paredes-Oliva), pbarlet@ac.upc.edu (Pere Barlet-Ros), fontas@tik.ee.ethz.ch (Xenofontas Dimitropoulos)

behavior, while new applications, services and cyber-threats are emerging. In this rapidly changing network environment, it is critical to build traffic profiling tools that efficiently process big traffic data to extract knowledge about what is happening in a network.

The most basic traffic profiling technique (and likely the most widely-used one) is extracting heavy hitter reports [1, 2, 3, 4] about top elements e.g., IP addresses that receive or generate most traffic. Finding a heavy hitter out of a given set of elements consists of identifying those elements with a frequency above a user-defined threshold. However, a simple traditional report of, for example, the source IP addresses consuming most traffic, does not give any information about what these hosts are actually doing, which reduces a lot the usefulness of such reports. To address this problem, previous work has studied the problem of finding multi-dimensional and hierarchical heavy hitters (HHHs) [5, 6, 7, 8, 9]. Notably, AutoFocus [10] is the best known tool for finding multi-dimensional HHHs. A HHH is an aggregate (e.g., an IP address prefix) on a hierarchy that appears frequently. The input data can be single- or multi-dimensional (e.g., IP address pairs) which gives rise to single- and multi-dimensional HHHs. An example multi-dimensional HHH provided by AutoFocus is <srcIP=*, dstIP=2.2.2.0/24, srcPort=80, dstPort=1024-65535, proto=TCP>. This HHH combines five different dimensions and takes the hierarchical nature of IP addresses into account. It highlights the pattern of many TCP responses from web servers (IP *, port 80) to IP addresses in the subnet 2.2.2.0/24.

Although AutoFocus is a state-of-the-art HHH-based traffic profiling tool, it has some important limitations. First, its computational complexity grows exponentially with the number of dimensions. This is because Autofocus requires n passes over the input data, where n is the total number of nodes in the multi-dimensional data structure that it builds to operate. This structure combines all the unidimensional hierarchies (one per dimension), into a multi-dimensional bigger hierarchy, i.e., each new dimension must be "replicated" along the already existing nodes for all the other dimensions. As a consequence, AutoFocus is restricted to 5-dimensional HHHs (where the five dimensions correspond to the source and destination IP addresses, the port numbers and the protocol) and it is very hard to extend it with additional traffic features. We would like to provide reports that contain other relevant information, such as the traffic application, the source and the destination ASes and the geographical location, so that we can provide a more precise and extensive summary of what is really happening in a network. For example, a report could look like <srcIP=*, dstIP=2.2.2.0/24, srcPort=80, dstPort=1024-65535, proto=TCP, srcGeo="Mountain View", srcAS=Google, dstGeo=Barcelona, dstAS=UPC, app=Youtube>. This 10-dimensional HHH reveals additionally that the web traffic was specifically exchanged between a particular subnet within the Barcelona campus of UPC and a particular Youtube data center of Google located in Mountain View.

Second, even with 5-dimensional HHHs, AutoFocus exhibits very high com-

putational complexity and cannot typically meet *near real-time*[1] guarantees even at low input rates. In particular, AutoFocus can process at most 10,000 flows/sec on commodity hardware, while commercial monitoring systems need to handle much higher rates.

We introduce a fundamentally new approach to extract HHHs based on generalized frequent item-set mining (FIM). In contrast to AutoFocus, which uses a straightforward approach that is not optimized for computational complexity [10] and leads to many passes over the data and high computational requirements, we build our proposal on state-of-the-art FIM algorithms that have been extensively optimized over the years. While adding new dimensions to AutoFocus requires to replicate each new dimension for all nodes, in FIM it simply consists of increasing the transaction length by one (as it will be described later, a transaction represents each of the entries in the input data, i.e., a set of items). Generalized FIM scales much better to higher dimensional data than AutoFocus and supports attributes of hierarchical nature, like IP addresses and geolocation. We exploit FIM to design and implement a new system, called *FaRNet* (FAst Recognition of high multi-dimensional NETwork traffic patterns), for (near) real-time profiling of network traffic data, which we first briefly described *FaRNet* in [11]. Our system is capable of analyzing multi-dimensional traffic records with both flat and hierarchical attributes.

We thoroughly evaluate the performance of *FaRNet* using real traffic traces from the European backbone network of GÉANT [12] and show that it scales very well to analyzing multi-dimensional data. We find that on commodity hardware that *FaRNet* can process up to 416,000 flows/sec with flat attributes and up to 127,500 flows/sec with hierarchical attributes. We also show that sampling can drastically reduce the memory consumption and CPU usage, while introducing only a very small error. Compared to AutoFocus, *FaRNet* is much faster, scales better to higher dimensions and produces traffic reports that are more meaningful for a network operator.

FaRNet has been deployed and used operationally for more than six months in the NOC of GÉANT. We report experiences on how generalized FIM is useful in practice.

In summary, we make the following contributions:

1. We introduce a new approach for discovering hierarchical multi-dimensional network traffic heavy hitters based on generalized frequent item-set mining.
2. We build a new system for network traffic profiling and thoroughly evaluate its performance. We show that *FaRNet* is much faster than AutoFocus. With flat attributes *FaRNet* can process up to 416,000 flows/sec and with hierarchical attributes up to 127,500 flows/sec on commodity hardware. In addition, it scales better than AutoFocus to higher dimensional data.

[1] We define *near real-time* as the requirement of fully processing an x-minute interval of traffic data in no longer than x minutes, where x is typically a small constant, like a 5-minute window.

3. Our system has been deployed and used for more than six months in the NOC of GÉANT, the European backbone network. We describe our experience on how generalized FIM is useful in practice.

The rest of this paper is organized as follows. Section 2 describes the FIM algorithms we evaluate. In Section 3, we describe *FaRNet* and propose modifications to existing FIM algorithms to efficiently deal with flat and hierarchical network traffic data. Section 4 shows the results after evaluating *FaRNet* and validating it against the well-known AutoFocus tool. Afterwards, Section 5 reports on the deployment of a prototype version of *FaRNet* in a real backbone network. Finally, Section 6 discusses the related work and Section 7 concludes our paper.

2. Background

FIM mines efficiently an input set of transactions to discover frequent patterns. Each input transaction T consists of a set of l items $T = \{e_1, ..., e_l\}$. A frequent item-set is a set of items that occur in at least s transactions. The threshold s is called *minimum support* and can be defined either as an absolute number or as a fraction of the total number of transactions. A classical application of FIM is in market basket analysis for finding groups of products that are purchased frequently together.

The *downward-closure* property of FIM states that an item-set is frequent *iff* all its subsets are frequent. For this reason, all the subsets of a frequent item-set are also frequent and the output of FIM can include many redundant item-sets. To address this problem, maximal frequent item-set mining has been proposed. Maximal item-set mining only returns the longest frequent item-sets and discards all the redundant subset frequent item-sets.

In this work, we model each traffic flow as a transaction where the items correspond to different flow features. Based on the standard 5-tuple definition, a flow is the set of packets with common values in the following 5 fields: the protocol number and the source/destination IP addresses and ports. Flow features are different characteristics of a flow, like the source/destination IP addresses and ports and the protocol (e.g., as in AutoFocus). However, many other can also be considered, like the number of packets or bytes of a flow, its source/destination AS or the layer 7 application. By default, *FaRNet* takes as input transactions that consist of the source and destination IP addresses, the source and destination port numbers, the protocol number, the inferred application that generated the flow, the source and destination ASes, and the geolocation of the IP addresses. The input can be trivially extended by adding new features, e.g., like path quality metrics, or by removing features, which is needed to compare the performance of *FaRNet* with AutoFocus. Each input attribute corresponds to one dimension in HHH terminology. Therefore, by default *FaRNet* processes 10-dimensional data.

In order to select the best performing algorithms for network traffic data, we evaluate five FIM algorithms. We select Apriori [13], FP-growth [14], and

Eclat [15] because they are the reference algorithms of the three main paradigms for computing a FIM solution. In addition, we select RElim [16] and SaM [17], which are two highly-optimized variants of FP-growth. Given a set of input transactions composed by items, FIM algorithms initially count the frequency of each single element. Afterwards, they combine these elements and iteratively discover sets of elements that frequently occur together. Different algorithms explore alternative and more efficient approaches for storing and combining items. The most well-known FIM algorithm (i.e., Apriori [13]) systematically generates combinations of all (single) items and then goes through the input transactions to check which combinations exist and are frequent, i.e., above the minimum support. More efficient algorithms (e.g., FP-growth [14]) are faster because they pass over the input transactions only twice instead of several times. Moreover, instead of combining items for all the dataset at every step, which generates a huge number of combinations, only a specific and much smaller set of items is used at each iteration. Next, we summarize the key features of each algorithm we have evaluated in this paper, i.e., Apriori, FP-growth, Eclat, Relim and SaM (further details can be found in [18]).

Apriori [13] is the first and simplest FIM algorithm. It works in breadth first order by iteratively merging frequent item-sets of increasing length. It starts by computing frequent item-sets of length one. Based on the *downward-closure* property, it then joins them to compute candidate item-sets of length two. Afterwards, it makes a pass over the input transactions and discards candidate item-sets that are not frequent. This procedure is repeated recursively until no more candidate item-sets can be generated. Apriori has two main drawbacks. First, it needs k passes over the input data, where k is the length of the longest item-set. Secondly, candidate generation and testing is extremely slow as the number of candidate item-sets can be very large. In order to overcome these issues, faster and more efficient algorithms have been proposed.

Eclat [15], instead of working with the typical horizontal representation of transactions, i.e., a list of items for each transaction, it uses a vertical layout, i.e., each item has an associated list of transaction identifiers where it appears in. It traverses the data in a depth first order and intersects the lists of transaction identifiers of the corresponding items for the counting.

FP-growth [14] uses a structure called Frequent Pattern Tree (FP-tree). In an FP-tree, those transactions sharing items will also share the same branch in the tree, which allows storing the data with higher compactness (especially for dense datasets). FP-growth skips the process of checking all candidate item-sets against all the database for each iteration, which is an extremely slow process and becomes intractable as input data increases or the *minimum support* gets lower. FP-growth improves this by significantly reducing the possible candidates to that part of the database related to a particular item (called the *conditional pattern base*). Another factor that explains why FP-growth is faster than Apriori is that it only reads the database twice.

RElim [16] is based on FP-growth although it does not use Frequent Pattern Trees. Instead, it proceeds by recursively eliminating items. First, it selects all transactions that have the least frequent item (among those items that are

	sIP	dIP	sP	dP	pr	app	sAS	dAS	sG	dG
Flow-based	1.1.1.1	2.2.2.2	5000	80	6	n/a	n/a	n/a	n/a	n/a
	1.1.1.2	2.3.3.3	6000	80	6	n/a	n/a	n/a	n/a	n/a
	1.1.1.3	2.4.4.4	7000	80	6	n/a	n/a	n/a	n/a	n/a
AutoFocus	1.1.1.0/30	2.0.0.0/8	hp	80	6	n/a	n/a	n/a	n/a	n/a
FaRNet	1.1.1.0/30	2.0.0.0/8	hp	80	6	HTTP	X	Y	US	EU

Table 1: Example of a traditional flow-based report (top three rows) and its equivalent reports for AutoFocus (middle) and *FaRNet* (bottom). sIP, dIP, sP, dP, sAS, dAS, sG and dG stand for the source and the destination IPs, ports, ASes, and geolocation, respectively. pr denotes the protocol and hp represents the high-ports, i.e., port values above 1023.

frequent). Then, it removes that item from them and recursively processes all the items left in that set of transactions, i.e., a dataset that is much smaller than the original. By remembering the items found during this recursion, when there are no more items left in the reduced dataset to be explored, RElim is able to compute all frequent item-sets associated with the removed item for which the recursion started. Afterwards, the algorithm repeats the process with the next least frequent item and without the already processed item in the database.

SaM [17], purely based on horizontal representation, is a simplified version of RElim. It performs in two steps: split and merge. In the split step, all arrays starting with the leading item of the first transaction are copied into new arrays and that leading item is removed. This process is repeated recursively to find all frequent item-sets for the leading item. Then, in order to obtain the conditional pattern base not containing the leading item, a merge step with the rest of the database not containing that item is needed. Optimized versions of both RElim and SaM have been recently proposed by the authors [19].

3. FaRNet: Building an efficient FIM system for network traffic

This section presents our system for FAst Recognition of high multi-dimensional NETwork traffic patterns (*FaRNet*). *FaRNet* efficiently analyzes network data and extracts useful frequent patterns that summarize the traffic activity of the network. First, Section 3.1 presents the architecture of *FaRNet*. Afterwards, we explain how we extend and optimize the FIM algorithms from Section 2 to deal with flat (Section 3.2) and hierarchical data (Section 3.3). Finally, Section 3.4 describes how we use sampling to improve the performance of *FaRNet*.

3.1. FaRNet Overview and Architecture

Even though previous works (e.g., HH, HHH) [5, 6, 7, 8, 9] are extremely useful to find frequent traffic patterns in network traffic, they are limited to explore a pre-defined set of dimensions (e.g., the well-known 5-tuple in case of AutoFocus [10]). Adding more dimensions results in an explosion in terms of runtime because of the exponential growth of generated combinations. The main objective of *FaRNet* is building a better tool that 1) is capable of dealing with high multi-dimensional data; 2) provides more comprehensive traffic reports; and 3) can perform in a timely fashion.

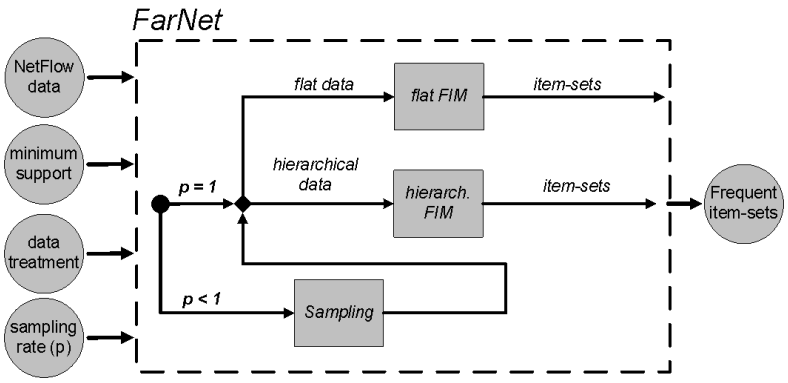

Figure 1: *FaRNet* architecture.

Figure 1 shows the architecture of *FaRNet*. As we can observe, *FaRNet* receives four inputs: NetFlow data, *minimum support (s)*, data treatment and sampling rate (p). s is the threshold that determines if the size of a set of flows is big enough to be considered a frequent item-set. The next parameter indicates the type of mining: flat or hierarchical. Elements that are treated as plain data are considered indivisible items, i.e., scalar values such as individual IP addresses or ports without considering their intrinsic structure. On the contrary, in the hierarchical scenario, elements are part of an associated hierarchy. For example, IP addresses consist of prefixes from length 8 to 32 and ports have a two-level hierarchy (specific port and its group, i.e., well-known or not). Similarly, the applications have also two elements, the specific application and its group (e.g., BitTorrent an P2P). Finally, geolocation data has four different elements for each IP (continent, country, region and city). For traffic classification, i.e., to infer the application that generated the flow, we used the technique described in [20], which uses machine learning and Sampled NetFlow data and obtains accurate results. First, there is an offline phase where the relation between a pre-defined set of traffic features (e.g., *port numbers* or *flow sizes*) and each application is analyzed. Afterwards, this set of features is used to build a classifier using machine learning. Finally, the created model is used to identify the network traffic online. Note that although our current implementation of *FaRNet* is based on these features, any other hierarchical element could be trivially added as the system scales well with the number of dimensions. Finally, the sampling rate parameter (p, $0 < p \leq 1$) indicates what percentage of the input data will be sampled. *FaRNet* has a single output: frequent item-sets.

FaRNet's first step is to sample the input NetFlow according to the sampling rate r introduced by the user. If $p = 1$, i.e., if all traffic is taken, no sampling process is actually needed and, therefore, it jumps directly to FIM. Otherwise, i.e., if $p < 1$, a certain percentage of the traffic will be randomly sampled before running FIM. Afterwards, depending on the selected type of mining, different paths will be taken. For flat treatment, a FIM algorithm for flat data will be

used. For hierarchical treatment, an optimized FIM algorithm extended to deal with hierarchical traffic attributes will be run. This extension, called *Progressive Expansion k-by-k (PEK)*, is explained in Section 3.3.3. The FIM boxes of Figure 1 perform maximal item-set mining to suppress redundant information. We adapt, extend and optimize the implementations of different FIM algorithms by [21] to deal with network traffic data. In particular, these implementations are designed to treat all input items equally. Consequently, we first adapt them to identify different data types, e.g., an IP address is not processed like the protocol number since the former is a hierarchical element and the latter is not. Second, the extension and optimization phases refer to our proposal, *PEK*, for efficiently dealing with hierarchical data.

In Table 1, we can observe several examples of network traffic activity reports. The first three lines represent a traditional flow-based report that describes the activity of different hosts accessing distinct servers on port 80. Line 4 shows the output that AutoFocus would return. As we can observe, this report offers more detail by showing the specific prefixes of source and destination IPs as well as the specific range of source ports used. Finally, line 5 shows the output of *FaRNet*. We can see that, additionally to what AutoFocus reported, *FaRNet* is able to find other interesting associations regarding the application used (column 6), the destination AS (column 8) and the geolocation (city, region, country and continent) for both the source and the destination IP addresses (last two columns). From the point of view of a network operator, *FaRNet* offers a much richer summarization of what is happening in a network w.r.t. to the extensive flow-based reports or the limited 5-tuple view offered by AutoFocus. As illustrated in Table 1, *FaRNet* is able to identify all flows as HTTP and also identifies the ASes and the physical source and destination of the communication, which is much more interesting and useful for the operator than only the 5-tuple or the raw set of flows.

3.2. FIM with flat attributes

With flat attributes, each flow record corresponds to a transaction with a fixed size of 10 items corresponding to the well-known 5-tuple (source and destination IP addresses, source and destination ports and protocol), the application, the source and destination AS and the source and destination data for the geolocation. All items are interpreted as plain data. In this case, *FaRNet* uses the FIM algorithms described in Section 2 with no further modification. In Section 4.2.1, we compare and evaluate the performance of these FIM algorithms with flat traffic data.

3.3. FIM with hierarchical attributes

Why is hierarchical mining interesting? For instance, suppose there is a high-volume horizontal scan towards a certain subnet. Although FIM flat would spot the attack, it would only report its source IP address and destination port. On the other hand, hierarchical FIM would find the specific subnet under attack and also discover if a certain range of ports was used (e.g., well-known

ports). AutoFocus would also find this type of information. Suppose now that there is a large amount of users from different countries in Europe massively accessing websites located in Asia. In this case, neither FIM flat nor AutoFocus would be able to discover this pattern. The former because it does not support hierarchies and the latter because it is limited to the 5-tuple. Nonetheless, FIM hierarchical would find this association in a higher level of the hierarchy, i.e., it would report interaction between Asia and Europe. In hierarchical FIM, IPs, ports, applications and geolocation data are treated as hierarchical elements.

In this paper, we propose the following three approaches to extend FIM to deal with hierarchical network traffic data: *Full Expansion* (Section 3.3.1), *Progressive Expansion* (Section 3.3.2) and *Progressive Expansion k-by-k* (Section 3.3.3). Next, we present the specific working scheme for each case.

3.3.1. Full Expansion

The straightforward solution for allowing FIM to deal with the hierarchical nature of network traffic is expanding each item of the transaction with its corresponding ancestors. For IPs, this means replacing each IP by all its possible prefixes from length 8 to 32, i.e., 25 items (note that prefixes of length shorter than 8 have not been considered since they have never been assigned). Regarding ports, they are translated into two items: its value and its corresponding range. If it is lower than 1024, it belongs to the well-known or low-ports group (0-1023). Otherwise ($\geq$ 1024), it is part of the high-ports (1024-65535). The protocol remains as a plain attribute. Similarly to ports, applications are also translated into two items: the specific application (e.g., Skype) and the group it belongs to (e.g., VoIP). Concerning the geolocation, for each IP address, we obtain four new elements: the continent, the country, the region and the city. Consequently, in *Full Expansion (FE)* all transactions are extended from 10 items (flat case) to 67 (25 items per IP, two items per port, one for the protocol, two for the application, one for each AS and four for the geolocation of each IP).

Hierarchical FIM will be able to provide greater granularity by automatically finding frequent IP prefixes, interaction between port ranges, associations between continents, countries, regions and cities and the applications generating such traffic. *Full Expansion* results are presented in Section 4.2.2.

3.3.2. Progressive Expansion

In *Full Expansion*, all transactions are always extended to 67 items when mining the 10 dimensions. Nonetheless, in most cases, extending all prefixes of an IP is not necessary because a fully defined 32-bit IP is rarely frequent by itself. On the contrary, prefixes of inferior length have higher chances of being above the *minimum support*. For instance, a certain prefix $a.b.0.0/16$ might be frequent even though a more specific subnet (e.g., $a.b.c.0/24$) is not. Similarly, while a city is not often frequent by itself, its corresponding region, country or continent might be.

For simplicity, from here on, all the extensions and optimizations will refer only to IP addresses. However, note that all the proposals made are applicable

Algorithm 1: Progressive Expansion k-by-k

```
Input: k: number of bits to expand at each step;
       ms: minimum support;
       Trees: array of trees for /8 prefixes;
1  for l = 8; l ≤ 32; l = l + k do
2      for every transaction T do
3          for every IP address e in T do
4              if l == 8 or parent_is_frequent(e,l - k,ms) then
5                  /*initialize a node if not set*/;
6                  n = get_node_by_prefix(Trees,e,l);
7                  n.count += T.weight;
8                  /*T.weight captures number of bytes, pkts, or flows*/;
9  /*compute unknown frequencies recursively*/;
10 for every tree TR in Trees do
11     if TR.child[left] not NULL then
12         TR.child[left].count = compute_frequencies(TR.child[left]);
13     if TR.child[right] not NULL then
14         TR.child[right].count = compute_frequencies(TR.child[right]);
15 for every transaction T do
16     for every IP address e in T do
17         expand_item(e,Trees,ms);
18         /*walk tree and return frequent ancestors*/
```

to the other hierarchical features presented in this paper and, in general, to any other hierarchical element.

Taking into account that an IP address is rarely frequent by itself, *Progressive Expansion (PE)* will not always generate all its 25 prefixes. An IP prefix of length k will only be explored if its corresponding $k-1$ prefix (parent prefix or ancestor) is frequent. Otherwise, the expansion for that IP will end at level $k-1$. This is because if a certain prefix is not frequent, all prefixes of superior length will not be frequent either (*downward-closure* property, Section 2). The frequency of a particular prefix is calculated by progressively counting the frequency of its shorter prefixes (see example in Section 3.3.3). *Progressive Expansion* results are reported in Section 4.2.3.

3.3.3. Progressive Expansion k-by-k

The main drawback of *PE* is that it needs to go through all transactions 25 times[2], which is very costly in terms of runtime. Consequently, this section presents *Progressive Expansion k-by-k (PEK)*, which seeks to reduce this part of

[2]Note that this is because of the depth of the IP address hierarchy and, therefore, it would change depending on the hierarchical element we are dealing with (e.g., 4 for the geolocation).

Algorithm 2: compute_frequencies

```
Input: n: node of the tree;
1 if n == NULL then
2     return 0;
3 else if n.level == 32 then
4     return n.count;
5 else
6     n.count = compute_frequencies(n.child[left]) +
      compute_frequencies(n.child[right]);
7     return n.count;
```

the process while avoiding the generation of useless prefixes. This is achieved by expanding k bits at each step instead of going one by one (*PE* is a particular case of *PEK* with $k = 1$). When using *PEK*, all transactions will be read $1 + 24/k$ times instead of 25. Note that the only valid values for k are 1, 2, 3, 4, 6, 8, 12 and 24.

Algorithm 1 shows *PEK*'s working scheme. First, all prefixes of length $l = 8$ are generated for all IPs of all transactions and, uniquely for these that are frequent, a binary tree is created (only the root node). Afterwards, for each prefix of length $l+k$ with a frequent ancestor (prefix of length l, tree level $l-8$), its corresponding tree is expanded up to level $l + k - 8$ (line 6). After going through all possible values of l ($8 \leq l \leq 32$), all frequencies in intermediate nodes (nodes between explored levels, i.e., among $l - 8$ and $l + k - 8$) are recursively computed (lines 12 and 14). Details of the recursivity can be found in Algorithm 2. Finally, transactions are expanded only with those prefixes that are known to be frequent by going though the corresponding tree from the root to the leaves following a depth-first approach (line 17).

The following example illustrates how *PEK* computes the prefixes and frequencies for the IP address 192.168.10.5 and $k = 2$. The first step consists of generating the binary tree for its prefix of length 8, i.e., 192/8. Afterwards, if the root node is frequent, prefixes of length 10 are generated (2-bit expansion). Therefore, frequencies for prefixes 192.192/10, 192.128/10, 192.64/10 and 192.0/10 are calculated. Then, the computation for intermediate nodes (prefixes of length 9) is calculated by moving backwards in the binary tree. In this case, prefixes 192.192/10 and 192.128/10 have a common ancestor, i.e., 192.128/9. Thus, the frequency of the intermediate node 192.128/9 is the sum of frequencies of its two descendants, 192.192/10 and 192.128/10. Likewise, the frequency for 192.0/9 comes from 192.64/10 and 192.0/10.

For $k = 24$, *PEK* would generate all prefixes of length 8 and 32 (same behavior as *FE* except for the first pruning of infrequent /8 prefixes). In this case, *PEK* would be extremely fast (it would go through all transactions only twice). However, it would use a lot of memory (it would directly expand all trees to the maximum of 25 levels without performing any sort of prefix pruning). There-

fore, our goal is finding a value of k such that the trade-off between memory usage and runtime is optimal. This trade-off is investigated in Section 4.2.4.

3.4. Sampling

In order to reduce the volume of input transactions, we can apply *random sampling*, i.e., we randomly sample each input record with probability p, where $0 < p \leq 1$. Sampling reduces the volume of input data and therefore speeds up the mining process. However, sampling can have undesired effects. In particular, these frequent item-sets obtained from sampled input may differ from those obtained from the full traffic. We identify the following four cases:

1. *Identical item-sets.* Both the sampled and the original output yield an identical frequent item-set that has exactly the same items. These flows matching such item-sets are *true positives*.
2. *Lost item-set.* An item-set that was in the original output but does not appear after applying sampling because it is undersampled and its frequency is not above the *minimum support* anymore. These flows belonging to such item-sets are *false negatives*.
3. *New item-set.* An item-set that is not frequent is oversampled and becomes frequent in the sampled transactions. Item-sets with frequency close to the *minimum support* are more likely to transition to frequent. Flows in this set are *false positives*.
4. *Transformed item-set.* This happens when two or more item-sets in the original output are merged into a new item-set in the sampled output. Normally, this new item-set has more defined items than the original because it is precisely a combination of them. In general, all the flows matching this item-set are *true positives* because even though the item-set is not strictly the same, its pattern defines the same set of flows matching the union of original item-sets.

Note that frequent item-sets close to the *minimum support (s)* are more likely to trigger false positives or negatives. The lower s, the higher the number of item-sets with frequency close to it. Therefore, the chances of either losing frequent item-sets or creating new ones are higher for smaller values of s.

To estimate the error due to sampling, we model the process with a binomial distribution. In particular, if N is the original size of an item-set, n the size of an item-set after sampling, and p the sampling rate, then the binomial distribution gives the probability of obtaining exactly n successes, i.e., a frequent item-set of size n, out of N independent trials, each of which yields a certain probability of success p. Using the binomial distribution, we can answer the following question. *Given a minimum support s, what reduced s' should be used in order to ensure with high probability that the original frequent item-sets will remain after applying sampling?* Our goal is to find what *minimum support* s' is needed in order to guarantee with a certain probability that an original item-set of N flows will be kept after sampling. Note that in order to keep a realistic view of

label	#flows	#packets	#bytes	duration
trace-1	0.51M	5.46M	5.55G	15m.
trace-2	1.96M	25.22M	21.88G	1h.
trace-3	3.87M	47.77M	42.20G	2h.
trace-4	5.77M	72.37M	67.83G	3h.

Table 2: Details of the *dataset*

what is happening in a network, our primary goal is to ensure that these item-sets that were originally frequent are also frequent after the sampling process. In exchange for that, we accept that some false positives might appear in the output. However, recall that false positives are item-sets that were very close to the minimum support in the unsampled scenario and, therefore, adding them in the final summary does not introduce any important error with respect to what is truly happening in the network. Nonetheless, missing frequent item-sets in the sampled output would impact significantly its usefulness. Being x the size of that item-set after sampling:

$$P(x > s') = 1 - P(x \leq s') = 1 - CDF(s', N, p) = 1 - \sum_{i=0}^{s'} \binom{N}{i} p^i (1-p)^{N-i} \quad (1)$$

From Eq. 1, the probability p of an item-set of 1000 flows of reaching $s = 100$ flows after applying a 10% sampling does not even reach 50% ($p \approx 47\%$). However, by using the binomial distribution we can find in advance a desirable setting. In this example, setting $s' = 60$ ensures that the original frequent item-set will not be lost. Nonetheless, there is no need to reduce the *minimum support* so aggressively. Using $s' = 80$ would allow keeping all the original item-sets with high probability ($p \approx 98\%$).

The impact of sampling on the performance and accuracy of *FaRNet* and the trade-off between *true positives* and *false positives* depending on the choice of s' will be discussed in Section 4.2.6.

4. Performance Evaluation

In this section, we first describe the scenario and datasets used in the evaluation (Section 4.1). Afterwards, we report *FaRNet* results with flat and hierarchical data, and also evaluate its performance under sampling (Section 4.2). Note that this part of the evaluation focuses on a simplified version of *FaRNet* with a limited number of dimensions (5-tuple) to allow the comparison with Auto-Focus [10] (Section 4.3). Finally, Section 4.4 reports on the results obtained by *FaRNet* when mining the full set of 10 dimensions (5-tuple, application, source/destination AS and source/destination geolocation).

4.1. Scenario and Datasets

The experiments presented in this section were performed using four NetFlow traces from 2011 from one of the 18 the points-of-presence of the European

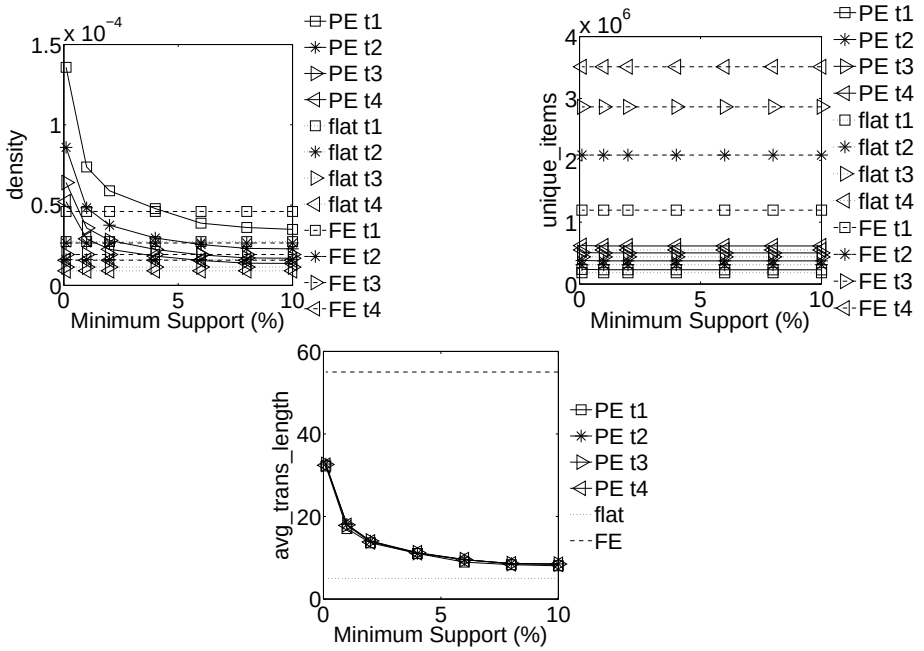

Figure 2: Density (left), number of unique items (middle) and average transaction length (right) for varying *minimum support* on *trace-1 (t1)*, *trace-2 (t2)*, *trace-3 (t3)* and *trace-4 (t4)*.

backbone network of GÉANT [12]. GÉANT is a /19 transit network connecting 34 European NRENs, a dozen of non-European NRENs and two commercial providers (Telia an Global Crossing). The details of the datasets can be found in Table 2.

For each traffic trace, Figure 2 reports the average transaction length (l), the number of unique items (n) and their density ($d = l/n$) for varying *minimum support* (s). As it will be discussed later, these parameters have an important impact on the performance of the FIM algorithms, and their values depend not only on the dataset, but also on the particular method applied; i.e., *flat*, *Full Expansion* (FE) or *Progressive Expansion (PE)*. For example, while in the flat case e.g., an IP address accounts for a single item, in *FE* each IP is always expanded to 25 items. Thus, both mechanisms lead to different dataset densities as they have different transaction lengths (l) and a distinct number of unique items (n). Nonetheless, for *PE*, these parameters also depend on s. This is because, as explained in Section 3.3.2, only specific prefixes are expanded in *PE* depending on s. Consequently, although the dataset is the same, for *PE*, both l and n change for different s. Specifically, the lower the value of s, the longer the transactions ($l \approx 8$ items for the highest s and $l \approx 32$ for the lowest). The figure confirms that network traffic data is extremely sparse (only datasets with $d \geq 0.1$ are considered dense in the literature [17]).

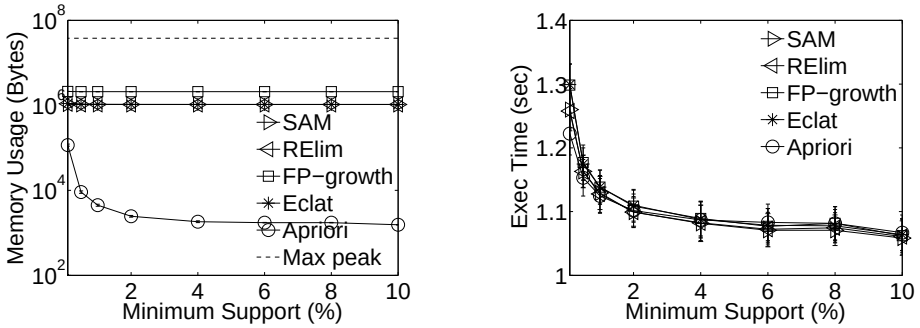

Figure 3: Memory usage (left) and execution time (right) with flat attributes and varying *minimum support* on *dataset*.

4.2. FaRNet Performance

In this section, we compare the algorithms presented in Section 2 with flat and hierarchical attributes. Afterwards, we select the best performing algorithm among them and show the achieved gain due to the optimizations proposed in Section 3. Finally, we discuss on the selection of the appropriate value of s. Recall that for all the experiments in this section, only the 5-tuple (i.e., source/destination IP addresses, source/destination ports and protocol) has been used.

We split the four NetFlow traces described in Table 2 in time bins of 15 minutes and show the average and the standard deviation in the results. From here on, in this section we will refer to that set of bins as the *dataset*. Note that the longer the time bin considered, the higher the resources needed (i.e., more running time and memory usage). We use an Intel Core2 Quad processor Q9300 and 3GB of main memory running Debian 5.0.5 (32 bits).

4.2.1. Flat treatment

Figure 3 shows the performance of all the evaluated FIM algorithms for flat data (i.e., *Apriori*, *Eclat*, *FP-growth*, *RElim* and *SAM*). On the left plot, we observe the memory used for values of s ranging from 0.1% to 10% (we discuss later in Section 4.2.7 on the appropriate selection of s). In order to clearly observe how each algorithm managed memory, the plots show two different values: the maximum total memory used ('Max peak') and the memory strictly reserved during the mining task. The main problem of reporting only the maximum peak is that the specific behavior of each method is completely hidden below that peak. In particular, the dotted line labeled as 'Max peak' in Figure 3 shows the maximum total memory used, which is the same for all the algorithms regardless of s ($\approx 36MB$). This happens for two reasons. First, because the part of the code that takes care of reading the input data and performing a first pass for counting the support of single element item-sets is common to all methods in the code provided in [21]. And second, because this first pass needs far more memory than the rest of the mining process.

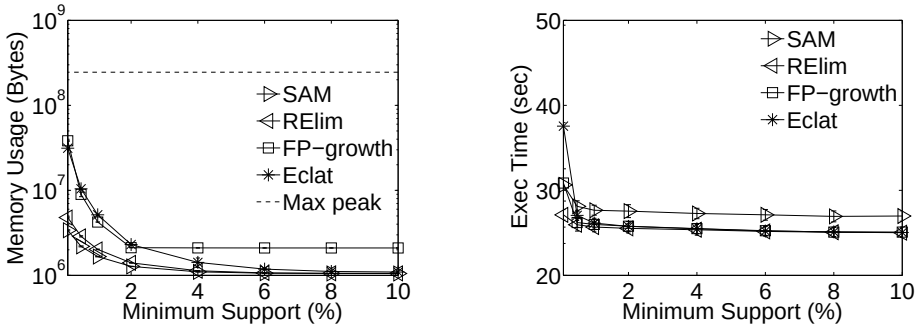

Figure 4: Memory usage (left) and execution time (right) with *Full Expansion* for varying *minimum support* on *dataset*.

Regarding the memory usage of each algorithm, *Apriori* is clearly the one with the lowest consumption. *FP-growth* shows the worst memory usage while *SAM, RElim* and *Eclat* report similar results. The main reason behind such memory differences between *Apriori* and the rest lies on the fact that, except for *Apriori*, all algorithms use complex data structures that book big memory blocks (e.g., ≈1 MB for *SAM*). On the other hand, *Apriori* asks for small pieces of memory every time it needs them.

In contrast, when looking at the execution time of each algorithm (Figure 3, right), the differences are hardly noticeable among each other. Overall, after analyzing both the runtime and the memory used for the mining by each of the algorithms, *Apriori* turns out to be the best algorithm for *FaRNet* for dealing with flat data.

4.2.2. Full Expansion

The objective of this section is to analyze how the presented FIM algorithms perform on hierarchical data to, later on, apply the optimizations presented in Section 3 to the best performing method. Recall that the goal of mining hierarchical data is to obtain higher granularity on the reported frequent itemsets. For example, if there is a scan, it might be possible to find out the specific subnet that is under attack (with flat treatment this is not possible).

Figure 4 shows the performance of *SAM, RElim, FP-growth* and *Eclat* when using *Full Expansion (FE)* on *dataset*. On the left plot, we can observe the memory consumption (in logarithmic scale) for different s values. *FP-growth* and *Eclat* show the highest memory consumption. However, while *FP-growth* is the worst performing algorithm for $s > 2\%$, *Eclat*'s memory consumption becomes close to *SAM*'s and *RElim*'s for $s \geq 4\%$. In contrast, both *SAM* and *RElim* scale smoothly for decreasing values of s (e.g., the memory usage for the mining is approximately one order of magnitude lower than *FP-growth*'s and *Eclat*'s for $s = 0.1\%$).

As regards the runtime, *RElim* and *FP-growth* are the fastest and, for $s < 1\%$, *RElim* is slightly better than *FP-growth*. *FP-growth* is among the quickest

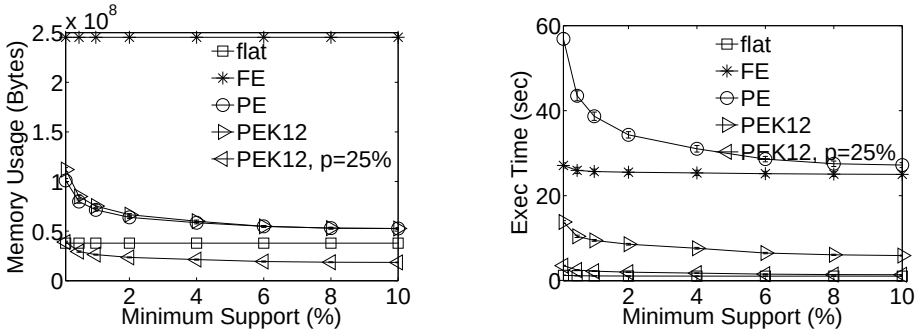

Figure 5: Memory usage (left) and execution time (right) comparison between *flat*, *Full Expansion*, *Progressive Expansion* and *Progressive Expansion k-by-k* for varying *minimum support* on *dataset*.

algorithms due to its compact FP-tree representation and *RElim* is particularly designed to deal with sparse datasets, which is the case for network traffic data (see Figure 2). *Eclat* performs similarly to *RElim* and *FP-growth* but only for $s \geq 1\%$. *SAM* turns out to be the slowest algorithm except for $s = 0.1\%$, in which case *Eclat* shows the worst results.

Note that *Apriori* algorithm does not appear in Figure 4 due to scalability issues, which impeded us to run it with hierarchical attributes even for the greatest $s = 10\%$. The main problem with *Apriori* is the candidate generation and testing step, which becomes too slow and needs too much memory with long transactions. Note that in this version of *FaRNet* limited to 5 dimensions (5-tuple), transactions have 55 items in *FE* instead of 5 as in the flat case (25 items for each IP, two per port and one for the protocol).

Based on Figure 4, *RElim* shows the best trade-off between execution time and memory consumption for *Full Expansion* in *FaRNet*. Therefore, in the following subsections we evaluate the different optimizations presented in Section 3 on top of *RElim*.

4.2.3. Progressive Expansion

In this section, we discuss the advantages of using *Progressive Expansion* over *Full Expansion*. Figure 5 shows the performance of *RElim* for all proposed methods: flat, *Full Expansion* (FE), *Progressive Expansion* (PE) and *Progressive Expansion k-by-k* (PEK). On the memory side (left plot), the effect of applying *PE* is clear. The figure reports the total memory consumption, including both the first pass to compute 1-element item-sets and the rest of the mining process. Note that unlike in Sections 4.2.1 and 4.2.2, the reported memory accounts for the total consumed, not only the part necessary for the mining tasks. This is because in previous sections we were analyzing what algorithm was performing better but from now on, we will progressively apply several improvements on the same algorithm, *RElim*. Therefore, we are not interested anymore in the memory consumed only during the mining but on the overall. With *PE*, the

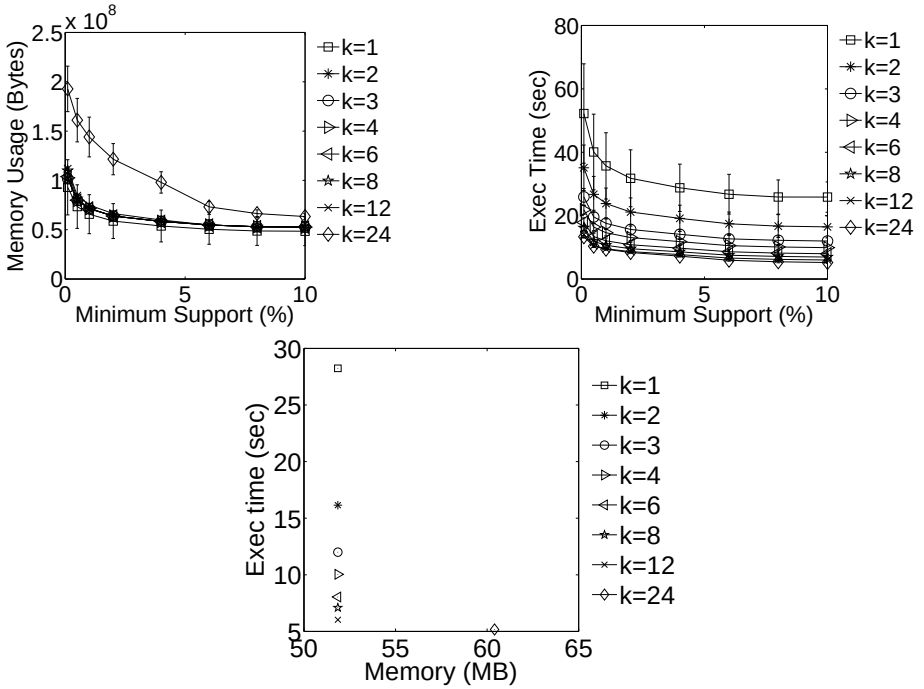

Figure 6: Results for *Progressive Expansion k-by-k* on *dataset*. Memory usage (left) and execution time (right) for varying *minimum support*. Memory-time trade-off for *minimum support = 10%* (bottom).

number of generated prefixes is dramatically reduced compared to *FE*, because of the fact that full IPs are rarely frequent enough to be above s and, therefore, they are generally cut before reaching /32. Consequently, the memory needed is far lower (e.g., $\approx 60\%$ reduction for the lowest s on *trace-1*). Moreover, as s increases, *PE* gets closer to flat treatment. In Figure 2 (middle), we can also observe that the average transaction length decreases with s. In particular, for $s = 0.1\%$ the average length is ≈ 32 elements (32.23), while for $s = 10\%$ it hardly reaches 8 elements (8.06).

Nonetheless, the pruning task results in an important increase of the execution time. Figure 5 (right) shows that *PE* is slower than *FE*. This is due to the fact that *PE* must go through all transactions 25 times, each one for extending all prefixes by one bit until reaching /32 (in *FE* every transaction is read only once and each IP directly expanded into 25 items). The next section evaluates an optimization that addresses this issue.

4.2.4. Progressive Expansion k-by-k

Figure 6 plots the memory usage (left) and the execution time (right) for all possible values of k when using *Progressive Expansion k-by-k (PEK)*. In terms of memory usage, $k = 24$ is the worst option (it expands directly all trees and,

label	#transactions	#transaction_size	#unique_items	#bytes_fixed_per_IP
Scenario 1	100,000	5	262714	0
Scenario 2	100,000	5	262226	1
Scenario 3	100,000	5	164904	2
Scenario 4	100,000	5	63257	3

Table 3: Details of the 4 synthetic datasets created

therefore, needs a lot of memory), while all the other values of k behave much better and very similarly among each other. On the contrary, when switching to the runtime comparison, $k = 24$ turns out to be the best choice together with $k = 12$ while $k = 1$ is the worst (it needs to analyze all transactions 25 times). Clearly, $k = 12$ offers the optimal trade-off in our scenario as it is almost as fast as $k = 24$ but uses much less memory. Figure 6 (bottom) reports the execution time and memory usage for $s = 10\%$ (other values of *minimum support* show almost identical results). This plot further confirms that $k = 12$ offers the best balance between speed and memory usage in our dataset.

Figure 5 shows that *PEK* with $k = 12$ outperforms both *FE* and *PE* in terms of both execution time and memory consumption. Although *PEK* and *PE* use approximately the same memory, *PEK*'s execution time is vastly reduced with respect to *PE*'s (transactions are only read three times to generate /8, /20 and /32 prefixes).

4.2.5. PEK: finding the optimal k

Note that the selection of k depends on the specific characteristics of the dataset (Section 4.1) and, therefore, another value of k might offer better performance in a different scenario. Next, we show how the optimal value of k changes for different synthetic datasets that have diverse characteristics. These datasets are composed by 5-tuple transactions. The source and destination ports and the protocol were randomly generated. In order to clearly observe the impact of k on the performance of *FaRNet*, we manipulated the source and destination IPs, i.e., those elements in the dataset that have a higher impact in the resource consumption due to their deeper hierarchy. We considered four scenarios that increasingly reduced the randomness of the dataset: 1) all source and destination IPs generated randomly, 2) all source and destination IPs with the first byte fixed, 3) both IPs with the first and the second bye fixed and, finally 4) both IPs with only the last byte generated randomly and the other three fixed. For more details on the dataset, refer to Table 3.

The performance of *FaRNet* varies significantly depending on the dataset and the value of k. In particular, we observed that while the impact of k in the runtime is consistent over all scenarios (lower execution time for higher values of k), its effect on the memory consumption varies a lot among the four datasets (see Figure 7). Specifically, the 4th scenario (bottom right figure) already shows that, unlike in Figure 6, $k = 12$ is not the best option. While $k = 24$ is still the fastest choice, its memory usage is extremely similar to the other values of k, i.e., $k = 24$ offers the best overall trade-off in this scenario. This is due to the fact that all IPs are extremely similar as the only random byte is the last

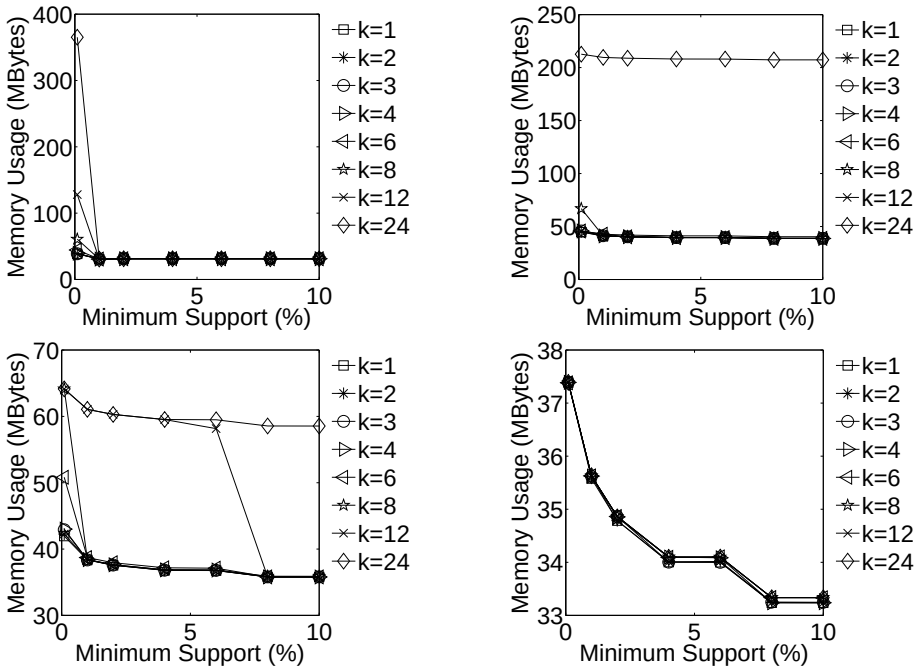

Figure 7: *FaRNet* memory usage for the four different artificial datasets generated: scenarios 1 and 2 (top left and right), and scenarios 3 and 4 (bottom left and right).

one. Consequently, k does not affect the memory usage because all IPs must be expanded almost fully. No pruning is possible for prefixes up to length 24, i.e., the 4th byte, as all IPs have frequent prefixes up to that length. As we increase the number of unique elements in the dataset (3rd scenario, bottom left plot), we find again that the optimal value of k changes. In particular, for high values of k and low s, the memory usage increases. The best trade-off in this scenario is $k = 6$ or $k = 8$ because of the higher sparsity of the dataset. Unlike the previous case, in this scenario *PEK* will rarely expand IPs beyond the 2nd byte. Therefore, $k = 12$ and $k = 24$ generate prefixes that will not be frequent. For this reason, lower values of k (e.g., 8 or 6) are better as they do not go further than the 2nd byte (i.e., they do not generate futile prefixes). Likewise, if we keep increasing the number of unique elements, (1st and 2nd scenario, top left and right plots), the previous phenomenon is further exacerbated, i.e., for higher k, more memory usage and lower execution time. This is because of the lower density of the datasets. In particular, the memory usage is up to 3 and 5 times higher than in previous scenarios for $k = 24$ and $s = 0.1$ (more unique elements to store, specially for low values of s) and the runtime is reduced by approximately half in scenario 1. The higher randomness in the data reduces a lot the number of frequent elements and, therefore, the mining is much simpler due to the lower number of elements to process after the pruning. Similarly to

the previous case, in these two last scenarios, the optimal choice is a small value of k.

4.2.6. Sampling

Next, we evaluate the impact of sampling on the output of *FaRNet*. We use the following metrics to quantify the error introduced by sampling. Let X be the union of flows matching all the frequent item-sets after running *FaRNet* on unsampled data. Likewise, Y defines this set of flows for the sampled case.

$$\text{True Positives Rate: } TPR = \frac{|X \cap Y|}{|X|}$$
$$\text{False Positives Rate: } FPR = \frac{|Y - X|}{|X|}$$

While *true positives* show the efficiency w.r.t. the unsampled case, the *false positives* indicate the correctness of the output (item-sets that were below the minimum support that turned frequent item-sets due to the sampling process). False positives are obtained by subtracting the set flows that belong to the original frequent item-sets detected without sampling (X) from set Y, which contains all the flows from all the frequent item-sets reported after the sampling process.

In Table 6, we can observe the error introduced in the output of *FaRNet* due to sampling for varying s and sampling rate p on *trace-1*. Concerning the TPR, we obtain a percentage close to 94% in the worst case. Moreover, for $p > 1\%$, the TPR is always extremely close to 100% regardless of s. We see that higher p and s result in smaller error. As regards the FPR, the worst results are clearly obtained for the lowest p. The highest value is 2.47%. However, for higher p, the number of false positives is greatly reduced, especially for low s. The reason why smaller values of s lead to less false positives is because the number of frequent item-sets that turn out to be mistakenly reported as frequent, account for a low number of flows w.r.t. the overall input flows.

For example, for $s = 10\%$ and $p = 1\%$, we obtain $TPR = 100\%$ and $FPR = 6.42\%$. These false positives are due to the fact that in the unsampled output, there is an item-set whose frequency is very close to s (9.96%) that, after sampling, depending on the case, becomes frequent (*new item-set* phenomenon described in Section 3.4). This only mistakenly classified item-set is provoking that peak in the FPR.

All in all, we can conclude that for $p = 10\%$ and above, we obtain great accuracy and low false positives. Also, for the most aggressive sampling rate, $p = 1\%$, we get acceptable values of both TPR and FPR as long as we use reasonably high values of s.

In Figure 5, we can observe the memory usage and runtime when applying sampling to *FaRNet*. As expected, when we apply a non-aggressive $p = 25\%$, the improvement in the performance is significant. Runtime beats clearly *PEK* w/o sampling and becomes very close to flat treatment. Regarding memory usage, it becomes the best among all methods, including the flat case. With this sampling rate, *FaRNet* needs less than 4 seconds in the worst case to process a trace of

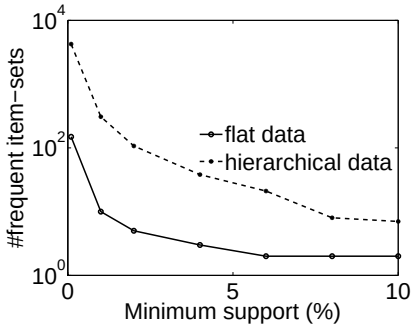

Figure 8: Number of frequent item-sets for flat and hierarchical data and varying *minimum support* for *trace-1*.

15 minutes with approximately half million flows (*trace-1*). This confirms that *FaRNet* performs (near) real-time.

4.2.7. Selecting the minimum support

Frequent item-sets are useful for many reasons (e.g., traffic profiling, anomaly extraction[22], anomaly detection [23]). They tell the operator what is happening in the network in a compact and summarized way. However, it is essential to tune the FIM system so that the amount of reported frequent item-sets is treatable by a human. While low s might lead to huge outputs with too much information, high s may hide interesting patterns. Therefore, finding the proper trade-off is crucial so that *FaRNet*'s output is useful for a network operator.

Figure 8 shows how the top-k of frequent item-sets varies for different s and data treatments (flat or hierarchical) when mining the 5-tuple on *trace-1* (see details in Table 2). While in the horizontal axis we see s from 0.1% to 10%, the vertical axis depicts the size of the output for a particular s (in logarithmic scale). We observe that for both flat and hierarchical attributes the number of item-sets decreases rapidly as s increases. However, the number of frequent item-sets changes significantly depending on how we treat the data. For flat attributes, the size of the output is always, at least, one order of magnitude lower than for hierarchical data, except for the most aggressive $s = 10\%$. This difference is due to the growth of frequent combinations between hierarchical elements, i.e., IP prefixes and ports. Therefore, in order to obtain a reasonable and humanly treatable number of item-sets in the output, the recommended s parameter changes significantly depending on the case.

Let us assume that we are interested in extracting the top-10 or top-20 of frequent item-sets. While we should use a $5\% \leq s \leq 10\%$ for hierarchical data, a s close to 1% would be more convenient for the flat case. For example, for *trace-1*, $s = 1\%$ returns 10 item-sets for flat treatment but, in order to reach a similar figure for the hierarchical case, s must be increased up to 8%.

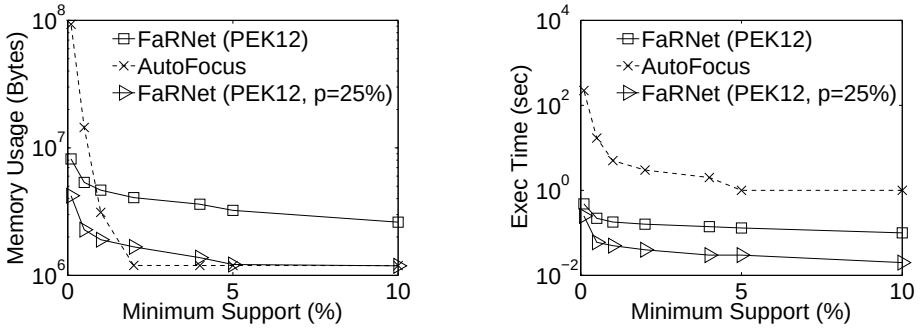

Figure 9: Memory usage (left) and execution time (right) comparison between *FaRNet* (*PEK12*) and AutoFocus for the first 10K flows of *trace-1*.

4.3. Comparison with AutoFocus

AutoFocus [10] is the only available tool of similar nature to our system (see Section 6 for details on AutoFocus). In order to validate the implementation of *FaRNet*, we compare it with AutoFocus. For the comparison, we configured both systems so that the same threshold is used to decide whether an item-set is frequent. While *FaRNet* uses the *minimum support (s)*, AutoFocus uses a parameter called *resolution*. We also limited the number of dimensions analyzed by *FaRNet* to only these attributes used by AutoFocus to perform the mining, i.e., five (the 5-tuple). For the experiments in this section, we used *trace-1* (see details in Table 2).

4.3.1. Performance comparison

Figure 9 shows how *FaRNet* and AutoFocus perform for different values of s. Note that only the first 10,000 flows of *trace-1* are used for this comparison. This is because the available implementation of AutoFocus [24] is not dimensioned to handle more flows (when it receives more than that amount, it does not count them accurately due to collisions). In terms of runtime (right plot), *FaRNet* is clearly faster regardless of s. Moreover, as s decreases, AutoFocus' execution time increases exponentially, while *FaRNet* is able to handle it smoothly. Although for the highest s AutoFocus' runtime ($1s$) is relatively close to *FaRNet*'s ($0.10s$), for $s = 0.1\%$ AutoFocus is approximately three orders of magnitude slower ($223s$ vs $0.48s$).

Regarding the memory consumption (left plot), AutoFocus is better than *FaRNet* for $s \geq 1\%$. However, for lower values of s, AutoFocus consumption rises rapidly and ends up consuming far more memory than *FaRNet* (88.77 MB vs 7.79 MB for $s = 0.1\%$). All in all, *FaRNet* shows to be quicker and more resilient to low s than AutoFocus, although it uses more memory for $s = 1\%$ and above. However, note that the memory consumption of *FaRNet* (without sampling) is reasonably low in the worst case (below 10 MB).

In order to improve *FaRNet* results, we apply sampling to the input data. In particular, the triangle dotted line in Figure 9 shows *FaRNet* under a 25%

sampling rate. As we can observe, *FaRNet* becomes faster than without sampling and, more importantly, gets really close to the memory consumption of AutoFocus. Under this sampling rate, *FaRNet*'s memory usage is lower than AutoFocus' for $s < 2\%$ and approximately the same for $s > 5\%$. However, for intermediate values of s (2% and 4%), *FaRNet* is still slightly worse.

4.3.2. Near real-time processing

An important advantage of *FaRNet* over AutoFocus is that *FaRNet* is capable of operating near real-time (i.e., it can process x minutes of data in less than x minutes). As shown in Figure 5, *FaRNet* without sampling can process 15 minutes of NetFlow data (*trace-1*) in significantly less than 20 seconds regardless of the *minimum support*. Note that *FaRNet* works on fixed time windows, i.e., it collects data during a certain time bin and when it ends it processes that data and produces results in less than the time bin. While processing, *FaRNet* needs to store part of the next time bin in a buffer, otherwise some traffic would be lost.

When applying 25% sampling, *FaRNet* can analyze *trace-1* with low error in less than 4 seconds in the worst case (lowest s). On the contrary, AutoFocus is not able to provide *near real-time results*. As depicted in Figure 9, for the first 10,000 flows of *trace-1* (which correspond to approximately 18 seconds of traffic), AutoFocus struggles for low s. For $s = 0.5\%$ it requires 16.25 seconds, which is slightly below near real-time, and for $s = 0.1\%$, AutoFocus needs almost 4 minutes, which is almost 12 times slower than real-time.

4.3.3. Scalability to more dimensions

The main issue of AutoFocus is its lack of scalability as the number of dimensions increases. Essentially, AutoFocus combines several unidimensional hierarchies (trees) into a multi-dimensional bigger hierarchy. The problem of such a pass is that the combinations grow exponentially with more dimensions. This is because each new dimension considered must be "replicated" for every single node of every unidimensional hierarchy. In particular, the number of clusters above the threshold is bounded by $r \prod_{i=1}^{k} d_i$, where k is the number of dimensions considered, i.e., 5 when mining the 5-tuple (i.e., srcIP, dstIP, srcPort, dstPort and protocol). d is the depth of the hierarchy of dimension i (e.g., 25 for IPs) and r is the resolution [10]. Therefore, the more dimensions and the higher the depth of the hierarchies, the worse. In contrast, adding more dimensions to *FaRNet* is as easy as increasing the transaction length. We have already seen that *FaRNet* easily handles transaction lengths of 55 items for the 5-tuple. In next section, we report results on analyzing the full set of 10 dimensions and show that *FaRNet* is able to process them very quickly.

4.4. Mining more dimensions

By mining the full set of 10 dimensions, item-sets like these reported in Table 7 can be obtained. In particular, this table shows the top-10 frequent item-sets reported by *FaRNet* on *trace-1* for $s = 1\%$. This top accounts for

approximately 10% of the traffic. The table also shows the item-sets returned by AutoFocus in order to observe the differences among both systems. *FaRNet* is able to process this trace (≈ 0.5 million flows) in approximately 12 seconds.

As we can see in the table, while AutoFocus essentially reports quite general IP prefixes, *FaRNet* is able to find much richer item-sets containing more concrete prefixes, specific locations, applications and ASes. For example, the 9th item-set reported by AutoFocus describes the activity of a a host that, judging by the source port, could be a web server. However, other than its source IP, the remaining data of the item-set is not extremely useful. However, when looking at the output provided by *FaRNet*, we can see that item-sets 1, 9 and 10 uncover some very interesting hidden associations related to the very same item-set. In particular, we confirm that the host is indeed a web server (its associated application is HTTP), we find its source AS and also discover that it is located in a city called Bethesda (Maryland, US). Specifically, this web server hosts a widely used tool in the research community that gives access to a database of references on life science and biomedical topics. Moreover, we observe that it is being accessed from several European countries (Great Britain, Germany and France) ans also find the corresponding destination AS for each case. Note that for the hierarchical attributes, only the most specific element is reported (e.g., in case of Bethesda, only the city is reported because adding the continent, the country and the region would be redundant). As we have been able to observe in this example, in order to understand what is really happening in the network, the item-sets reported by *FaRNet* are more synthetic and, therefore, much more informative than these returned by AutoFocus.

5. Deployment

A prototype version of *FaRNet* [25] based on flat treatment on the 5-tuple [22] has been deployed in the European-wide backbone network of GÉANT (see scenario details in Section 4.1). It is essentially used by the network operators for automatically extracting and summarizing in a compact view all the traffic flows causing an anomaly (*anomaly extraction*). Even though *FaRNet* is a tool for traffic profiling, our main intuition for using it for *anomaly extraction* is that anomalies often result in large sets of flows with similar characteristics (e.g., when there is a DDoS, lots of flows share the same destination IP and port). Consequently, anomalies will appear in the reports of *FaRNet* as frequent item-sets. In particular, *FaRNet* has been used for two purposes in GÉANT: 1) validate anomaly alarms previously triggered by an already existing anomaly detection tool in the network (*NetReflex* [26]) and 2) collect malicious evidence in a compact summary to send it to the involved parties.

The process of analyzing an anomaly works as follows: 1) *NetReflex* triggers an alarm and provides the related meta-data (e.g., involved IPs), which is used to extract a set of candidate flows responsible for the anomaly; 2) each flow is modeled as an item-set with 5 items and *FaRNet* is used to extract the frequent item-sets out of the large set of candidate flows obtained in step 1. *FaRNet* also provides the network operators with a GUI that allows them to extract the

sIP	dIP	sPort	dPort
X.191.64.165	Y.13.137.129	55548	*
Z.66.124.39	Y.13.137.129	*	*
*	Y.13.137.129	3072	80
*	Y.13.137.129	1024	80

Table 4: List of item-sets found by *FaRNet* for a vertical scan detected by *NetReflex*.

frequent item-sets associated with an alarm, investigate the raw flows matching any returned item-set, and tune the extraction parameters if needed (e.g., the *minimum support*).

In order to ensure that the prototype of *FaRNet* was working as expected, we performed an evaluation during the deployment process in order to validate its results. During this evaluation, we randomly selected anomalies from 10 days. For 42% of the anomalies, *FaRNet* found uniquely the item-set strictly related to the meta-data reported by *NetReflex*. Additionally, for 26% of the cases, the algorithm evidenced additional flows related to the anomaly that were not provided by the anomaly detector. These were particularly interesting cases because *FaRNet* was able to discover new anomalies that had been missed by *NetReflex*. For example, the following meta-data were signaled and labeled as a vertical scan by *NetReflex*: "sIP: X.191.64.165, dIP: Y.13.137.129, srcPort: 55548 and dstPort: *". When analyzing the same anomaly using *FaRNet*, the frequent item-sets in Table 4 were found. The 1st was precisely the item-set responsible of the anomaly already flagged by *NetReflex*. The 2nd was another host doing a similar vertical scan on the same target, while the 3rd and 4th were two simultaneous DDoS on port 80 against the same target. We believe that this capability of finding more flows related to an anomaly has general applicability. Moreover, in 26% of the cases, some additional item-sets related to legitimate activity were extracted, which could be trivially filtered out by the network operator. For the remaining 6% of the alarms, *FaRNet* was not able to extract meaningful flows, which could be due to a stealthy anomaly not captured by our extraction technique or due to a false positive-alarm (FP).

Overall, *FaRNet* was extremely useful for the network operators of GÉANT. In particular, it provided the following:

1. Fast and more reliable anomaly analysis w.r.t. the time-consuming and error-prone manual investigation.
2. Discovery of additional information related to an anomaly that was missed by *NetReflex*.
3. Easier identification of FP reported by *NetReflex*.
4. Useful and compact summaries of the traffic.

6. Related Work

A large number of FIM algorithms have been studied in the literature. A survey of existing algorithms can be found in [18]. More recently, researchers

Related Work	Hierarchical Items	Real-Time	Dimensionality
FIM	√	-	High
HHH	√	√	Low
AutoFocus	√	-	Low
FaRNet	√	√	High

Table 5: Taxonomy of the related work

have studied the related problem of finding hierarchical heavy hitters (HHH) [5, 6, 7, 8, 9, 1, 2, 3, 4]. Given a stream of items, e.g., IP addresses, a HHH is an aggregate, e.g., an IP address prefix, on a hierarchy that appears often. The HHH problem is, in fact, a special case of the more general FIM problem. HHH algorithms typically process the input data in a streaming fashion, approximate the HHHs, and can accommodate a small number of dimensions. Most FIM algorithms operate in an offline fashion, i.e., making multiple passes over input data, without approximation and scale better to a large number of dimensions. Furthermore, finding heavy hitters over streams of flat (network traffic) attributes is a widely-studied special case of the more general HHH problem (and in turn of the FIM problem). Heavy hitters are simply frequent items in an 1-dimensional stream of flat items.

AutoFocus [10] is a well-known system for finding HHHs over network traffic data. In contrast, to most other HHH algorithms, it operates in an offline manner making multiple passes over the input data. It takes as input 5-tuples of IP addresses, ports, and protocol, it treats IP addresses and ports as hierarchical attributes, and it finds frequent 5-dimensional aggregates. AutoFocus is the most related previous work. In fact, AutoFocus is essentially a particular case of *FaRNet*. The main contribution of our work is to build a new traffic profiling system based on FIM principles that is much faster, more flexible and more general than AutoFocus and allows to easily extend the input 5-tuples to include a much larger number of dimensions.

In Table 5 we summarize how the main previous works differ in the dimensionality of the input records, the type of items (flat or hierarchical), and the (near) real-time or offline processing of the input records.

7. Conclusions

In this paper, we analyzed the performance of state-of-the-art FIM algorithms when applied to network traffic data, and extended and optimized them to deal with hierarchical dimensions such as IP addresses, ports, applications and geolocation data. We also evaluated the impact of sampling on the performance of FIM algorithms, and showed that it significantly reduces both execution time and memory usage while providing precise output.

Based on this analysis, we built *FaRNet*, a network traffic profiling system that offers better performance and flexibility and also scales to a much higher number of dimensions than AutoFocus. In order to validate the correctness of *FaRNet*, we compare it with AutoFocus by using a limited version of our system configured to produce the same output. Using traffic data from a large backbone

network, we show that when mining only the 5-tuple, *FaRNet* is able to process 15 minutes of traffic in less than 4 seconds with a very small error. We show that *FaRNet* is up to three orders of magnitude quicker than AutoFocus. As a consequence, *FaRNet* is able to process high volumes of multi-dimensional traffic data in (near) real-time, while AutoFocus was designed for offline analysis of a pre-defined set of 5 dimensions. Finally, when analyzing the full set of 10 dimensions, *FaRNet* confirmed its ability to produce more useful and synthetic reports and also showed that it scales very well with the number of dimensions by analyzing 15 minutes of traffic in approximately 12 seconds (w/o sampling).

We deployed a preliminary version of *FaRNet*, demonstrated in [25], in the European-wide backbone network of GÉANT and showed its usefulness and good results for assisting network engineers when dealing with anomalies in an operational environment.

Acknowledgements

We thank DANTE for having provided us the GÉANT traffic traces. This work was partially supported by the TMA-COST Action IC0703, the Spanish Ministry of Education under contract TEC2011-27474 and the Catalan Government under contract 2009SGR-1140.

	$p = 1\%$		$p = 10\%$		$p = 25\%$		$p = 50\%$	
	TPR (%)	FPR (%)	TPR (%)	FPR (%)	TPR (%)	FPR (%)	TPR (%)	FPR (%)
$s = 0.1\%$	93.64 ± 0.32	1.38 ± 0.12	98.76 ± 0.16	0.9 ± 0.09	99.26 ± 0.15	0.55 ± 0.1	99.6 ± 0.1	0.35 ± 0.06
$s = 1\%$	97.26 ± 0.65	1.77 ± 0.45	98.87 ± 0.18	0.78 ± 0.22	99.33 ± 0.41	0.36 ± 0.09	99.19 ± 0.25	0.24 ± 0.11
$s = 2\%$	96.81 ± 1.62	1.24 ± 0.65	99.72 ± 0.27	0.12 ± 0.04	99.96 ± 0.03	0.11 ± 0.04	99.99 ± 0.03	0.06 ± 0.04
$s = 4\%$	95.76 ± 5.22	0.46 ± 0.36	98.33 ± 1.62	0.31 ± 0.28	99.36 ± 1.34	0.11 ± 0.21	99.12 ± 1.44	0.11 ± 0.21
$s = 6\%$	98.73 ± 1.2	2.47 ± 2.86	99.75 ± 0.26	1.46 ± 2.59	99.78 ± 0.26	0.23 ± 0.32	99.88 ± 0.19	0.35 ± 0.32
$s = 8\%$	98.94 ± 1.43	0.41 ± 0.76	100 ± 0	0 ± 0	100 ± 0	0 ± 0	100 ± 0	0 ± 0
$s = 10\%$	100 ± 0	1.32 ± 2.88	100 ± 0	1.29 ± 2.89	100 ± 0	0 ± 0	100 ± 0	0 ± 0

Table 6: Impact of sampling on *FaRNet* for *trace-1*: $mean \pm stdev$ for true positives rate (TPR) and false positives rate (FPR) for different *minimum supports (s)* and sampling rates (p).

	item-set	sIP	dIP	sPort	dPort	proto	app	sAS	dAS	sGeoloc	dGeoloc
AutoFocus	1	*	*	lp	hp	6	n/a	n/a	n/a	n/a	n/a
	2	*	*	80	hp	6	n/a	n/a	n/a	n/a	n/a
	3	*	*	hp	hp	6	n/a	n/a	n/a	n/a	n/a
	4	A.0.0.0/8	*	lp	hp	6	n/a	n/a	n/a	n/a	n/a
	5	*	*	hp	lp	6	n/a	n/a	n/a	n/a	n/a
	6	A.0.0.0/10	*	*	hp	6	n/a	n/a	n/a	n/a	n/a
	7	A.0.0.0/10	*	lp	hp	*	n/a	n/a	n/a	n/a	n/a
	8	A.0.0.0/9	*	80	hp	6	n/a	n/a	n/a	n/a	n/a
	9	A.14.29.110/32	*	80	hp	6	n/a	n/a	n/a	n/a	n/a
	10	*	*	hp	80	6	n/a	n/a	n/a	n/a	n/a
FaRNet	1	A.14.29.110/32	*	80	hp	6	HTTP	70	786	Bethesda	GB
	2	*	B.128.0.0/10	80	hp	6	HTTP	*	*	US	EU
	3	C.66.122.0/24	*	80	hp	6	HTTP	32	*	Stanford	EU
	4	C.66.120.0/21	*	80	hp	6	HTTP	32	786	Stanford	GB
	5	*	B.48.0.0/13	80	hp	6	HTTP	*	2200	US	FR
	6	D.234.254.12/30	*	80	hp	6	HTTP	6932	*	Danvers	EU
	7	E.100.0.0/18	*	hp	hp	6	*	1103	*	Amsterdam	*
	8	D.234.252.14/32	*	80	hp	6	HTTP	6932	*	Danvers	EU
	9	A.14.29.110/32	*	80	hp	6	HTTP	70	680	Bethesda	DE
	10	A.14.29.110/32	*	80	hp	6	HTTP	70	2200	Bethesda	FR

Table 7: Top-10 frequent item-sets reported by AutoFocus and *FaRNet* on *trace-1* sorted by descending frequency (*s=1%*). lp and hp stand for low-ports and high-ports, respectively.

[1] C. Estan, G. Varghese, New directions in traffic measurement and accounting, in: Proc. of ACM SIGCOMM, 2002.

[2] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, F. True, Deriving traffic demands for operational IP networks: methodology and experience, in: Proc. of ACM SIGCOMM, 2000.

[3] B. Babcock, C. Olston, Distributed top-k monitoring, in: Proc. of ACM SIGMOD, 2003.

[4] A. Metwally, D. Agrawal, A. El Abbadi, Efficient computation of frequent and top-k elements in data streams, in: Proc. of ICDT, 2005.

[5] G. Cormode, F. Korn, S. Muthukrishnan, D. Srivastava, Finding hierarchical heavy hitters in data streams, in: Proc. of VLDB, 2003.

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

[7] G. Cormode, F. Korn, S. Muthukrishnan, D. Srivastava, Finding hierarchical heavy hitters in streaming data, ACM TKDD 1 (4) (2008) 2:1–2:48.

[8] J. Hershberger, N. Shrivastava, S. Suri, C. Tóth, Space complexity of hierarchical heavy hitters in multi-dimensional data streams, in: Proc. of ACM SIGMOD/PODS, 2005.

[9] Y. Zhang, S. Singh, S. Sen, N. Duffield, C. Lund, Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications, in: Proc. of ACM IMC, 2004.

[10] C. Estan, S. Savage, G. Varghese, Automatically inferring patterns of resource consumption in network traffic, ACM CCR 33 (4) (2003) 137–150.

[11] I. Paredes-Oliva, P. Barlet-Ros, X. Dimitropoulos, FaRNet: Fast Recognition of High Multi-Dimensional Network Traffic Patterns, in: Proc.of ACM SIGMETRICS (poster), 2013.

[12] GÉANT, http://www.geant.net.

[13] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proc. of VLDB, 1994.

[14] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, ACM SIGMOD Record 29 (2) (2000) 1–12.

[15] M. Zaki, S. Parthasarathy, M. Ogihara, W. Li, et al., New algorithms for fast discovery of association rules, in: Proc. of KDD, 1997.

[16] C. Borgelt, Keeping things simple: finding frequent item sets by recursive elimination, in: Proc. of OSDM, 2005.

[17] C. Borgelt, X. Wang, Sam: A split and merge algorithm for fuzzy frequent item set mining, in: Proc. of IFSA/EUSFLAT, 2009.

[18] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, Data Min. Knowl. Discov. 15 (1) (2007) 55–86.

[19] C. Borgelt, Simple algorithms for frequent item set mining, Advances in Machine Learning II 263 (2010) 351–369.

[20] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, J. Solé-Pareta, Analysis of the impact of sampling on netflow traffic classification, Computer Networks 55 (5) (2011) 1083–1099.

[21] Christian Borgelt's Software, http://www.borgelt.net/software.html.

[22] D. Brauckhoff, X. Dimitropoulos, A. Wagner, K. Salamatian, Anomaly extraction in backbone networks using association rules, in: Proc. of ACM IMC, 2009.

[23] I. Paredes-Oliva, I. Castell-Uroz, P. Barlet-Ros, X. Dimitropoulos, J. Solé-Pareta, Practical Anomaly Detection based on Classifying Frequent Traffic Patterns, in: Proc. of IEEE GI, 2012.

[24] AutoFocus implementation, http://www.caida.org/tools/.

[25] I. Paredes-Oliva, X. Dimitropoulos, M. Molina, P. Barlet-Ros, D. Brauckhoff, Automating Root-Cause Analysis of Network Anomalies using Frequent Itemset Mining, in: Proc.of ACM SIGCOMM (demo), 2010.

[26] Guavus, NetReflex, http://www.guavus.com.