Seek and Push: Detecting Large Traffic Aggregates in the Dataplane

Jan Kuˇcera1, Diana Andreea Popescu2, Gianni Antichi2, Jan Koˇrenek3, Andrew W. Moore2

1CESNET, CZ 2University of Cambridge, UK 3Brno University of Technology, CZ

# Abstract

arXiv:1805.05993v1 [cs.NI] 15 May 2018

High level goals such as bandwidth provisioning, ac- counting and network anomaly detection can be easily met if high-volume traffic clusters are detected in real time. This paper presents Elastic Trie, an alternative to approaches leveraging controller-dataplane architec- tures.

Our solution is a novel push-based network monitor- ing approach that allows detection, within the data- plane, of high-volume traffic clusters. Notifications from the switch to the controller can be sent only as required, avoiding the transmission or processing of unnecessary data. Furthermore, the dataplane can iteratively re- fine the responsible IP prefixes allowing a controller to receive a flexible granularity information. We report and discuss an evaluation of our P4-based prototype, showing our solution to be able to detect (with 95% of precision), hierarchical heavy hitters and superspread- ers using less than 8KB or 80KB of active memory re- spectively. Finally, Elastic Trie can identify changes in the network traffic patterns, symptomatic of Denial-of- Service attack events.

# Introduction

The importance of finding high-volume traffic clusters has been recognized in the past to improve network man- agement practices [[23,](#_bookmark52) [20,](#_bookmark49) [29,](#_bookmark58) [37,](#_bookmark66) [48].](#_bookmark77) Specific applica- tions include, as shown in Table [1,](#_bookmark0) accounting [[21,](#_bookmark50) [23],](#_bookmark52) traffic engineering [[24,](#_bookmark53) [6],](#_bookmark35) anomaly detection [[33,](#_bookmark62) [34],](#_bookmark63) Distributed Denial-of-Service (DDoS), and scans detec- tion [[53,](#_bookmark82) [51].](#_bookmark80)

Dataplane monitoring is the main instrument that en- ables the detection of high-volume traffic clusters. In the past, it was based on packet sampling [[16,](#_bookmark45) [2],](#_bookmark31) to lower overheads and data collection bandwidth, thus impacting estimation accuracy [[14,](#_bookmark43) [22,](#_bookmark51) [38].](#_bookmark67) OpenFlow (OF) [[39]](#_bookmark68) did not improve the situation either [[20]:](#_bookmark49) the main monitoring mechanism exposes the per-port and per-flow counters available in the switches. An appli- cation running on top of the controller can periodically poll counters using the standard OF APIs, and then per-

form a software-based algorithm to get insights into the network behavior. However, this approach limits signif- icantly the original flexibility intended by Software De- fined Networking (SDN). While increasing the gap be- tween two consecutive counters requests reduces the con- troller ability to react in a timely fashion, continuously requesting counters from switches leads to non-scalable solutions by challenging the switch-controller interac- tions capabilities [[20].](#_bookmark49) For this reason, lately a num- ber of proposals suggest the use of programmable data- planes using the P4 programming language [[8]](#_bookmark37) to extend dataplane functionality with more advanced monitoring features. Specifically, some of them leverage dataplane programmability to either directly provide the top-k heavy hitters [[48]](#_bookmark77) or to assist the controller by export- ing smart representations of aggregated traffic statis- tics [[55,](#_bookmark84) [37].](#_bookmark66) While the latter case results in a more flexible and generic approach compared to the former, the controller still needs to receive at a fixed time inter- val the generated information from the dataplane, and estimate the various application-level metrics of interest based on the received data. Such an architecture might suffer from the same problem of the legacy OF proto- col: the ability to apply network policy updates based on the received data depends on the switch-controller’s interactions capabilities of collecting statistics at short time ranges [[20].](#_bookmark49)

In this paper we take a different approach. We lever- age dataplane programmability to transform the switch from a passive monitoring infrastructure to an *active*

|  |  |
| --- | --- |
| **Network event** | **Management task** |
| (Hierarchical)  Heavy Hitters | accounting, traffic engineering |
| Changes in  traffic patterns | anomaly detection, DoS detection |
| Superspreaders | worm, scan, DDoS detection |

**Table 1: Detecting high-volume traffic clusters is beneftcial for a number of network management tasks.**

system which is capable of detecting several types of net- work events associated with high-volume traffic clusters, and only subsequently to inform the controller. We de- signed a new data structure, *Elastic Trie*, that enables the detection of hierarchical heavy hitters, changes in network traffic and superspreaders from within the data- plane, and we present its implementation in P4. The ba- sic idea behind the proposed solution is to create a hash table based prefix tree that grows or collapses to focus only on the prefixes that account for a ”large enough” share of the traffic. This enables the detection of (hierar- chical) heavy hitters, and by looking at its growing rate it is possible to identify changes in the traffic patterns.

The main contributions of the paper are as follows:

We propose a *push-based* approach to network mon- itoring, where the dataplane informs the control plane only when specific conditions are met.

*•*

30

25

IBM RackSwitch G8264

NoviSwitch 1132

Average time (seconds)

20

15

10

5

0

0 10000 20000 30000 40000 50000 60000 70000 80000

Number of hardware counters

**Figure 1: Time to retrieve hardware counters.**

10

IBM RackSwitch G8264 NoviSwitch 1132

Average time (seconds)

9

8

7

6

5

4

3

2

1

0

We present a data structure that enables the de- tection of a number of network events associated with high-volume traffic clusters within the data- plane. Specifically, we demonstrate how Elastic Trie allows to detect hierarchical heavy hitters, changes in network traffic and superspreaders. Moreover, our solution iteratively refines the responsible pre- fixes so that the controller receives a finer or coarser grained information depending on the desired re- porting time.

*•*

We implemented our idea in P4 using match-action tables and we demonstrate its detection capabilities by evaluating it through trace-driven simulations.

*•*

# Motivating Event Triggered Monitoring

We first ran an experiment to measure the amount of time it takes to retrieve an increasing number of hard- ware counters from a switch. We used two different switches. The first is a fairly new OpenFlow-enabled IBM solution, i.e., RackSwitch G8264 [[35],](#_bookmark64) capable of

1.2 Tbps throughput. The second is the NoviSwitch 1132 [[42],](#_bookmark71) which has been designed for use in high band- width and flow-intensive network deployments. We con- nected the switches to a server running an OpenFlow controller and we built an application that allows to re- quest an increasing number of flow counters from the switches which were idle when the counters were pulled. Figure [1](#_bookmark1) shows the results we obtained. Surprisingly, the IBM switch reports values aligned with tests per- formed against much older solutions [[20],](#_bookmark49) while the No- viSwitch performs much better. We did not manage to perform our test for more than 100K counters, but

0 5000 10000 15000 20000 25000 30000

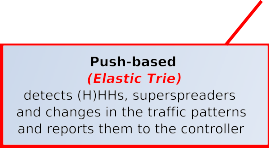
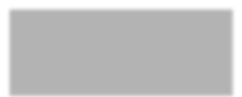
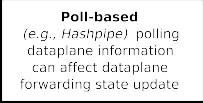
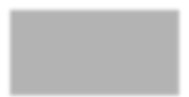
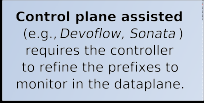
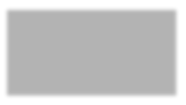
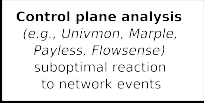
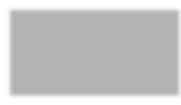
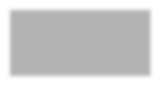
Number of rules

**Figure 2: Time to add new rules.**

the increasing trend of the figure for both switches is clear. Although dataplane programmability can help in reducing the number of counters exported by aggregat- ing flow rules in probabilistic data structures, such as bloom filters or sketches, past research has shown that around 150K counters are still required to provide use- ful information to the controller [[37].](#_bookmark66) While the IBM switch can take up to 28 seconds to retrieve approxi- mately half of the aforementioned amount of counters, the NoviSwitch needs at least 5 seconds. Certainly, such delays are not acceptable when it comes to critical net- work events detection, e.g., DoS, DDoS, scans, worms. The lesson learned is that retrieving a large number of flow counters from hardware is time consuming.

Using the same configuration as in the previous ex- periment, we ran the second test to characterize the amount of time it takes to modify an increasing number of rules on a high-end switch. Updating the forwarding state and retrieving statistics from a controller are two competing operations that are commonly performed se- quentially by the switch. The larger the number of flow additions or statistics requests, the bigger the impact of one action on the completion of the other [[20].](#_bookmark49) For this reason it is important to characterize switch rule update time, especially because issuing a large number of forwarding updates in a single batch is a common de- fense practice for Internet Service Providers (ISPs) to stop DDoS attacks, and can involve up to 50K rules update [[25].](#_bookmark54)

Figure [2](#_bookmark2) shows the results we obtained. The reported values show a lower bound of the switch update time. In the test we pushed an increasing number of rule modifi-



**Figure 3: Design space for detection of high volume traffic clusters.**

cations with the same priority. This is the best case for a switch, as demonstrated by past research [[27],](#_bookmark56) as TCAMs do not have to reorder the hardware rules. Adding 30K rules, in the best case scenario, might re- quire 10 seconds for the IBM switch and almost 4 sec- onds for the NoviSwitch. This would affect the statistics collection capabilities of the controller.

The results obtained motivated us to build a solution that: (1) does not depend on statistics retrieval from the dataplane, and (2) decouples the monitoring and the for- warding statistics updates. Indeed, with Elastic Trie the controller receives a push-based notification only when an event related to high-volume traffic clusters has been detected in the dataplane.

# Desired Properties

Figure [3](#_bookmark4) surveys the design space for the detection of high volume traffic clusters and places our solution, *Elastic Trie*, in context by following the thick red lines through the design tree. This section describes the in- sights that inform our major design decisions and pro- vides the necessary background for the selected network events. In designing Elastic Trie, we targeted a solution with a number of key features:

**Efficiency.** Collecting counter statistics from all the active flows or smart representations of aggregated traf- fic statistics [[55,](#_bookmark84) [37]](#_bookmark66) can create considerable control plane load. With Elastic Trie, we aim for a push-based solution which exports information to a controller only when a network event has been detected in the data- plane.

**Packet processing independence.** The main OpenFlow mechanism for dataplane monitoring exposes the per-port and per-flow counters available in the switches [[39].](#_bookmark68) Although this might seem a logical and simple solution, it suffers from a major drawback: ex- pensive TCAM resources must be shared between the

rules needed for packet processing and the rules installed for monitoring purposes only [[57].](#_bookmark86) In addition, the use of pre-configured monitoring rules requires prior knowl- edge of the active network flows, as well as a large num- ber of fine grained rules, in order to accurately detect heavy flows. With Elastic Trie, we aim to decouple the packet processing logic from the monitoring mechanism. While the former can still use the available TCAM re- sources, the latter can be implemented algorithmically with match-action tables using the P4 programming lan- guage.

**Historical network trend awareness.** Change de- tection is the process of identifying flows that contribute the most to traffic change over two consecutive time in- tervals [[12].](#_bookmark41) Previous solutions [[37,](#_bookmark66) [26]](#_bookmark55) rely on the con- troller to compute the differences from multiple inter- vals, effectively slowing down the reaction capability of the network if an anomaly has been found. With Elas- tic Trie, we seek a solution capable to directly compute such an operation within the dataplane at the expense of minimal memory consumption.

**Optimization for fast traffic steering.** Net- works today rely on middleboxes to provide security and added-value services [[46].](#_bookmark75) Taking advantage of global network knowledge of a controller, it is easy to enforce a network policy change and steer (part of) the traf- fic if an anomaly has been detected [[44,](#_bookmark73) [28].](#_bookmark57) Collecting statistics from the dataplane, running the detection al- gorithm in the control plane, and then enforcing a policy change if something suspicious is found can cause delays that might not be acceptable in the case of a network attack. With Elastic Trie, we propose to detect at short timescales a coarse-grained approximation of the prefix responsible for the network traffic changes. Once the detection of the anomalous subnet is done in the dat- aplane, the traffic can be instantaneously redirected to the appropriate middlebox, without the need to commu- nicate with the controller.

**Optimization for network management.** Divid- ing the time in fixed intervals can simplify the detec- tion of a number of network events, e.g., heavy hit- ter, superspreader, DDoS. At the end of each time win- dow, it is possible to identify the flows that consume more than a fraction *T* of the link capacity, i.e., heavy hitter, or determine the host that contacts more than a number of unique destinations, i.e., superspreader. For this reason, current solutions for network monitor- ing typically operate by exporting counters or specific data structures, e.g., sketches, to the controller at fixed time scales [[54,](#_bookmark83) [37,](#_bookmark66) [36].](#_bookmark65) However, this approach tightly bounds the reactive capabilities of the network with the dataplane statistics reporting time, as it needs to be (at least) comparable to traffic variations [[3,](#_bookmark32) [6].](#_bookmark35) Only if this last condition is met, solutions like dynamic routing of

heavy flows [[20,](#_bookmark49) [6,](#_bookmark35) [45]](#_bookmark74) or dynamic flow scheduling [[47]](#_bookmark76) can be easily implemented. However, state of the art so- lutions adopt a fairly large reporting time (typically 20 seconds [[37,](#_bookmark66) [48])](#_bookmark77) not to overload the controller with too much data, thus limiting network reaction capabilities. With Elastic Trie, we target a solution that iteratively refines the responsible prefixes in the dataplane. In this way, the controller, depending on the desired reporting time, can receive finer or coarser grained information on the flow responsible for a network event associated with a high-volume traffic cluster.

## Selected Network Events

This section provides the necessary background for the network events considered in this paper.

**(H)HH detection.** Hierarchical Heavy Hitters have already been studied in a number of prior works [[58,](#_bookmark87) [19,](#_bookmark48) [29,](#_bookmark58) [40,](#_bookmark69) [13,](#_bookmark42) [4].](#_bookmark33) Detecting an Heavy Hitter (HH) means identifying a large aggregate in the network traffic. For example, assuming the use of the source IP address as a key, the goal of the HH detection problem is to find the source IP prefixes that contribute with a traffic vol-

ume[1](#_bookmark7) larger than a given threshold *T* during a fixed time interval *t*. Figure [4](#_bookmark9) depicts the amount of traffic for prefixes in a reduced 3-bit wide model domain of IP

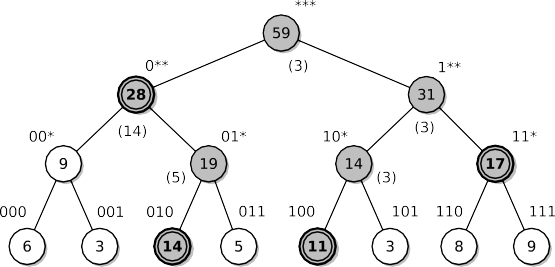
addresses. All of the prefixes are denoted as a prefix tree, also known as trie. Each node of the trie has at most two children. The left child is associated with bit value 0, while the right child is associated with bit value 1, and the prefix *p* represented by a node is defined by the path from the root to that node. Terminal nodes express only the traffic volume produced by full IP ad- dresses. Non-terminal nodes then summarize the traffic of a prefix. The contribution of each prefix is repre- sented as a number in each node. Considering the use of a threshold *T* = 10, terminal nodes 010, 100, non- terminal node 11\* and all their ancestors are identified as heavy hitters. For example, each child of the 11\* node contributes independently less than the threshold *T* , but in total both children contribute enough to ex- ceed the threshold and report the 11\* prefix as a HH.

A Hierarchical Heavy Hitter (HHH) [[19]](#_bookmark48) is a special case of HH. Specifically, it is a prefix *p*, which exceeds a threshold *T* after excluding the contribution of all its HHH descendants[2](#_bookmark8). In Figure [4,](#_bookmark9) only prefixes 010, 100, 0\*\* and 11\* are HHHs. The amount of traffic of each HH prefix without the impact of its HHH descendants is shown in brackets. In this example, the 11\* node is an HHH, as none of its children contributes enough to exceed the threshold *T* , but the amount of traffic from both children exceeds the threshold. In contrast, the

1It can be considered in terms of packets or bytes per second.

2The descendant prefixes need to satisfy the definition of HHH.

**Figure 4: A trie of IP addresses in reduced 3-bit model. Each node represents a preftx** *p* **with asso- ciated amount of traffic sent. Assuming threshold** *T* = 10**, grey nodes are heavy hitters, while double circle nodes are also hierarchical heavy hitters.**



1\*\* prefix is not an HHH because a significant part of its contribution originates from its descendant nodes 100 and 11\*, which are already HHH and must be excluded. It is worth noting that, while the detection of HHHs requires the knowledge of the HHs, the opposite is not true. Reporting to a controller the HHHs guarantees minimum overhead, while providing all the necessary information. Taking as an example the configuration of Figure [4,](#_bookmark9) a dataplane capable of detecting HHs would export to the controller the following prefixes: 0\*\*, 1\*\*, 01\*, 10\*, 11\*, 010 and 100. In contrast, a dat- aplane with HHH detection capability would report just 0\*\*, 11\*, 010 and 100. In both cases the amount of useful information for network management practices is

the same[3](#_bookmark10), but in the second case we export less data.

**Change detection.** Traffic anomalies are a normal

occurrence in the daily life of network operators. While some of them can be sometimes tolerable, others are of- ten an indication of performance bottlenecks due to flash crowds [[30],](#_bookmark59) network element failures, or malicious ac- tivities such as Denial-of-Service attacks (DoS), worms and spam. Change detection is one of the main ap- proach to network anomaly detection. The method de- tects traffic anomalies by deriving a model of normal behavior based on the past traffic history and looking for significant changes in short-term behavior that are inconsistent with the model [[32].](#_bookmark61) Identifying the flows responsible for the changes in the traffic patterns can be formulated also (at least in part) as a high-volume traffic clusters detection problem [[23].](#_bookmark52) Specifically, it re- quires the ability to discover which flows contribute the most to the traffic changes over two consecutive time intervals [[12].](#_bookmark41)

**Superspreader detection.** A superspreader is de- fined to be a host that contacts at least a given num- ber of distinct destinations over a short time period. It

3Some of the reported HHs are just prefixes of more specific HHs.

can be responsible for fast worm propagation, so detect- ing it early is of paramount importance [[52].](#_bookmark81) Moreover, superspreader detection can be seen as a high-volume traffic cluster identification problem. Specifically, while past examples, such as HH, typically define the traffic volume in terms of packets or bytes per second, in the case of superspreaders, the problem is tackled in the di- mension of flows per second. While an HH is a source that sends a lot of traffic, a superspreader is a source that contacts many distinct destinations. In addition, superspreader detection can be seen also as Distributed Denial of Service (DDoS) victim detection if, instead of the source, the same type of spread detection is applied to the destination [[55].](#_bookmark84)

# Elastic Trie Algorithm

The Elastic Trie algorithm is inspired by past works on HHH detection [[58,](#_bookmark87) [29].](#_bookmark58) Specifically, it operates in the same hierarchical manner. It also enables the de- tection of a number of network events associated with high-volume traffic clusters from within the dataplane without the need to be coordinated by a controller. Fi- nally, it operates in a packet-driven manner and can be implemented using common match-action based archi- tectures such as RMT [[9].](#_bookmark38)

In this section, for the sake of simplicity, we first intro- duce a basic version of the Elastic Trie which is capable to detect only HHHs. We then describe its mapping into appropriate P4 constructs, e.g., match-action tables and registers. Finally, we discuss extensions to the basic al- gorithm to support the detection of other events, such as superspreaders and network traffic changes.

## Data Structure & Basic Algorithm

Let us assume that we know in advance all the poten- tial HHHs in a network. In this case, to correctly de- tect which of them is a real HHH, for each new arriv- ing packet it is necessary to lookup the longest prefix matching (LPM) in the table of potential HHHs and then increment its associated counter. Thus, in some aspects, this is similar to the IP lookup problem [[49],](#_bookmark78) where the longest prefix matching in the forwarding ta- ble is searched. In practice, the two problems, while sharing some common aspects, are quite different. In the first case, the forwarding table is computed by the con- trol plane, does not directly depend on the nature of the dataplane traffic and does not change very frequently. In contrast, a table storing HHHs is very dynamic, as it is correlated with the properties of the dataplane traffic. In addition, since the HHH prefixes are not known in

The nature of the HHH problem (IP addresses can be naturally organized according to prefixes into a hierar- chy) led us to use a tree-based data structure. Thus, for the purpose of HHH detection we maintain a standard trie data structure [[49].](#_bookmark78) A trie is a tree data structure, where the position of a node in the tree defines the key associated with it. Every node in a trie has at most two child nodes[4](#_bookmark12). The left child is associated with bit value 0, and the right child is associated with bit value

1. Each node also represents a prefix, which is defined by the path from the root of the tree to that specific node. With Elastic Trie, we further extend this concept and associate a specific data structure within each node. Specifically, it consists of three elements: the counter as- sociated to the left child (32 bits), the one associated to the right child (32 bits) and a timestamp (48 bits). The counters represent the amount of traffic, e.g., pack- ets or bytes, for each of the node’s direct subprefix, while the sum of the counters represents the amount of traffic sent by the prefix itself. The timestamp specifies the time when the node was created or the last time when the counters were reset.

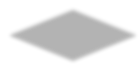
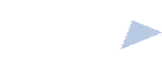
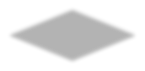
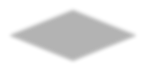
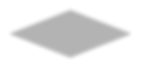
The starting condition is associated to a trie composed by a single node, corresponding with the zero-length pre- fix \*. The basic idea behind the proposed solution is to have a trie that grows or collapses to focus on the nodes associated to prefixes that account for a ”large enough” share of the traffic. Thus, we named our data structure Elastic Trie. To achieve this, inspired by the NetFlow [[16]](#_bookmark45) terminology, we defined two time intervals: *active timeout tA* and *inactive timeout tI* , where *tA < tI* . The active timeout *tA* is the interval after which the prefix is evaluated and possibly reported as HHH to the controller. The inactive timeout *tI* defines the interval after which the IP prefix corresponding to the node is considered inactive and its counters outdated. Figure [5](#_bookmark13) depicts key steps of the proposed Elastic Trie algorithm. For every incoming packet, the longest prefix (thus its corresponding node) is looked up and the packet times- tamp (*tP* ) compared against the node timestamp (*tN* ). Let us also denote by *c*0 and *c*1 the left and the right child counters of the found node. Here, there are five possible cases that have to be considered based on the result of the comparison, the node counter values, time- outs *tA* and *tI* :

**Invalidating the node.** If the inactive timeout *tI* expires (*tI tP tN* ), it means the prefix node has been inactive for a long time. The values of the coun- ters are outdated and are not relevant for the detec- tion any more. This can happen when the source prefix stops sending packets for a while. Because the detec- tion is built on a packet-driven basis this can not be de-

*≤ −*

advance, all the traffic received needs to be monitored

to properly build the corresponding HHHs table. 4For the sake of simplicity let us now ignore the multi-bit tries.



**Figure 5: Flowchart showcasing input packet pro- cessing of the Elastic Trie detection algorithm.**

tected easily in the dataplane. Thus, the inactive time- out mechanism helps to handle this situation when the packets of the source prefix start to flow again and when the old values must be invalidated. Figure [6a](#_bookmark16) illustrates this case. Regardless of the counter values, the tree node is simply removed and the counter values discarded.

**Expanding the node.** This is the case when both the active and inactive timeouts have not expired yet (*tP − tN < tA < tI* ), but one of the node counters (let

us assume, for example, *c*0) exceeds the threshold *T* that

the system uses to discriminate heavy prefixes (*c*0 *≥ T* ). In this case, the subprefix associated with *c*0 can be

(optionally) reported to the controller as HH but not as HHH yet. Figure [6b](#_bookmark16) depicts this case: the data structure automatically starts the refinement of the prefix (10\*) by creating a new child node (100) corresponding to *c*0.

According to the definition of HHH, the original *c*0 must

be set to zero to remove the contribution of the newly

created descendant prefix. Since, we also do not have any records for the newly created child yet, the new node will have its timestamp set to the current packet timestamp and both its counters set to zero.

**Keeping the node.** This is the case when the inac- tive timeout *tI* has not expired, but the active timeout *tA* has expired (*tA ≤ tP − tN < tI* ), and the sum of

*≥*

both counters exceeds the threshold *T* (*c*0 + *c*1 *T* ),

but none of the counters contributes enough to reach the threshold individually (*c*0 *< T* ; *c*1 *< T* ). This case is shown in Figure [6c.](#_bookmark16) When such a condition happens, the prefix corresponding to the node (11\*) is a HHH, be-

cause it exceeds the threshold *T* and none of its children contributes enough to exceed to threshold individually. The prefix then is reported to the controller, its times- tamp updated with the packet timestamp value and the counters are reset to prepare the node for the evaluation in next time interval.

**Collapsing the node.** If the inactive timeout *tI* has not expired yet, the active timeout *tA* has expired (*tA ≤ tP − tN < tI* ) and the sum of both counters does

not exceed the threshold *T* (*c*0 + *c*1 *< T* ), the node can

be collapsed. This case is depicted in Figure [6d.](#_bookmark16) The node (10\*) is removed from the tree structure, and it is replaced by the nearest parent. Both the counters of the parent node (1\*\*) are zeroed and the timestamp is set to the current packet timestamp. Also note the difference between collapsing and invalidation of the node. In the case of invalidation the nearest parent is not reinserted or renewed.

**Updating the node counter.** This is the only ac- tion which is performed when both the active and inac- tive timeouts have not expired yet (*tP tN < tA < tI* ) and none of the node counters exceed the threshold *T* (*c*0 *< T* ; *c*1 *< T* ). In this scenario, the node counter corresponding to the packet subprefix is updated[5](#_bookmark14) and the trie structure does not change. Note the counter is also updated after other actions when the node is kept, expanded or collapsed. In these cases the newly created node or the nearest parent node counters are updated instead of the current node counter.

*−*

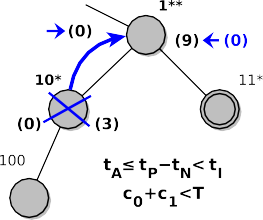
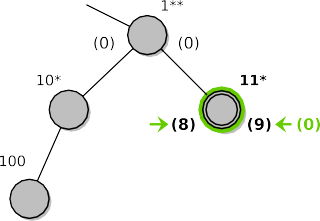
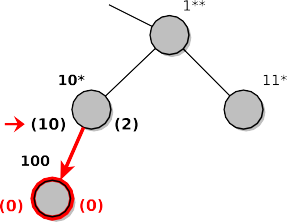
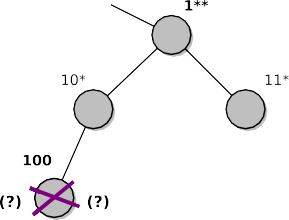
## Elastic Trie Prototype in P4

This section discusses the implementation of Elastic Trie on programmable hardware, using the P416 specification version 1.0.0 [[18].](#_bookmark47) Figure [7](#_bookmark17) depicts a high-level view of the architecture and illustrates the operations performed for each incoming packet. The structure is organized around three main building blocks: (A) the LPM clas- sification stage, (B) the main memory used to gather traffic statistics alongside related timestamps and (C) the control logic to dynamically adjust the hierarchical data structure and to report (partial) results of the de- tection to the external controller.

Each incoming packet is first parsed to extract the desired flow key, i.e., source IP[6](#_bookmark15). Then, the hierarchical tree structure is accessed to find the LPM (step 1). The result of this stage is an address that is used to access the main memory, where the data structure of the as- sociated node is stored (step 2). The reported values are thus compared with the packet timestamp (step 3) and the appropriate operation is computed (step 4) fol- lowing the Elastic Trie specifications described in the previous section. Specifically, the comparison can trig- ger an update of the main memory (step 4a), an update of the LPM classification scheme (step 4b), or a push notification to the external controller (step 4c). In the following, we provide a more detailed description of the mapping between the three main building blocks and *P* 416 match-action constructs.

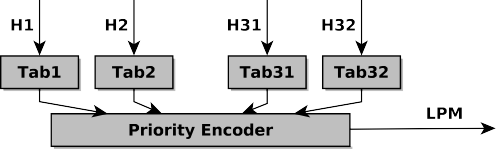
5Depending on the trie configuration, the counters might carry information about bytes or number of packets.

6While Elastic Trie is oblivious with respect to the specific packet field used as flow key, the source IP address is commonly used for the HH and HHHs detection.



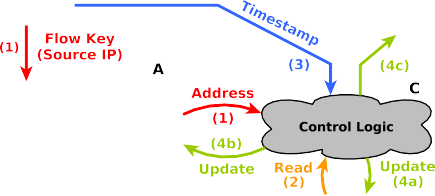
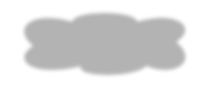
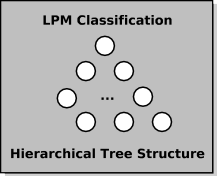
**(a) Invalidating the node. (b) Expanding the node. (c) Keeping the node. (d) Collapsing the node.**

**Figure 6: The core cases of Elastic Trie reftnement, assuming the threshold** *T* = 10**. Each node represents a preftx and builds the data structure. Node counters are shown in brackets on the sides.**



**Figure 8: The LPM classiftcation stage in P4.**

**Figure 7: Elastic Trie dataplane architecture.**



**LPM classiftcation stage.** Although P4 offers built-in match tables supporting LPM, we could not uti- lize them for implementation of the trie structure, since the latest P4 specification does not support modifica- tions of these tables directly from the dataplane, even though some targets like FPGAs may support it. As this feature is essential for our architecture, we opted for a custom LPM implementation. We use a hash ta- ble for each prefix length (Figure [8),](#_bookmark18) thus requiring 32 hash tables to support each IPv4 prefix[7](#_bookmark19). Each hash table is implemented as a register array. Upon packet arrival, all the hash tables are read in parallel, by hash- ing the associated prefix of the flow key. We use hash extern API with CRC32 as an algorithm to generate hash values to access the registers. Hash tables referring to short prefix values usually require less memory, as they need to store information for a smaller number of results. Thus, depending on the amount of memory allocated to each hash table, we use a direct access based only on the prefix value itself (so called IDENTITY hash algorithm in P4 API) for some of the shortest prefix tables. Each individual hash table lookup result can then be repre- sented as a single bit value, 1 (found), 0 (not found) respectively. Using bitwise operators we put these bits

7Using less hash tables and supporting only a subset of prefixes comes at the cost of node complexity. Indeed, each node needs to store a counter for each associated subprefix. This means that if we use only hash tables for just the prefixes 8, 16, 24 and 32, we need to construct nodes with 256 counters each.

*\ \ \ \*

together to form a bitvector, which serves as an input key into a static ternary match table that implements a structure similar to a priority encoder.

**Main memory access mechanic stage.** The hash value of the resulting LPM is used as address to access a register array that stores the required node structure information for that specific prefix, i.e., two packet coun- ters and a timestamp. We use 32-bit wide packet coun- ters and 48-bit wide timestamp as it is available in the packet metadata structure in P4. To detect hash colli- sions in our implementation of LPM classification stage in P4, we further extended the node data structure with a up to 32-bit wide flow key field (IPv4 prefix). Note that we do not need to store the prefix length because we use a separate hash table for each length. Thus, the size of each node structure is 144 bits (112 bits for the node and 32 bits for the IP address). Then, in the case of a hash collision, the nearest shorter prefix node is used instead of the reported LPM.

**Control logic.** This last stage compares the packet timestamp with the node timestamp and applies the logic described in Section [4.1.](#_bookmark11) The node collapse or ex- pansion is performed by updating the appropriate hash table storing the specific prefix that needs to be ad- justed, while the push-based mechanic is implemented by generating a packet digest (digest extern object in P4 API) containing the IP prefix detected as HH or HHH alongside its node information such as the sum of the counters and the timestamp. The controller does not directly participate on the trie refinement and receives only generated messages. Thus, for further evaluation

of the control logic and using an available API of P4 be- havioral model [[17]](#_bookmark46) we also implemented a lightweight command line tool in Python to receive and dump re- ported (H)HH prefixes.

## Enhanced algorithm

In this part we introduce three further extensions of the basic algorithm described in the previous section. First, we address the support for the detection of other network events, i.e., superspreaders and network traffic changes. Second, we provide an optimization that ac- celerates the trie building process, thus improving the reactiveness of our solution.

**Superspreaders detection.** As introduced in sec- tion [3.1,](#_bookmark6) a superspreader is defined to be a host that contacts at least a given number of distinct destinations over a short time period. Thus, to enable such a de- tection, it is important to keep track of the number of destinations contacted by each source prefix. To address this challenge we used a standard Bloom filter [[7],](#_bookmark36) which is a memory-efficient probabilistic data structure com- monly used to test for set membership. Specifically, we deployed the filter in parallel to the main memory to test if a packet belongs to a new unique flow or not. The key to index the filter consists of the source IP prefix looked up during LPM classification phase and destination IP address of the packet. The control logic to dynamically adjust the hierarchical structure is kept the same as in the basic algorithm, however, a test-and-set operation on the filter is performed for each incoming packet and the appropriate node counters are updated only if a new unique flow is detected. This change to the architecture allows the detection of (hierarchical) superspreaders us- ing the Elastic Trie data structure. Moreover, the Bloom filter can be easily implemented in P4 as a bit array placed in a register and a set of *k* hash functions.

**Change detection.** The common way to detect changes in the traffic patterns is based on deriving a model of normal behavior based on the past traffic his- tory and looking for significant changes in short-term be- havior that are inconsistent with the model itself. One of the desired properties for Elastic Trie was to be histori- cal network trend aware (Section [3).](#_bookmark5) Indeed, by tracking the number of nodes expanded or collapsed over an ac- tive timeout interval *tA*, it is possible to spot sudden changes. To enable such a detection, we added a global timestamp register and an integer counter which is in- cremented and decremented when any node of the tree is expanded or collapsed, respectively. When the traffic is steady, the number of nodes expanded and collapsed should be similar and the value of the counter should

nificant change in short-term traffic behavior which is reported to the controller using a digest message.

**Variable active timeout.** The starting condition for our data structure is associated to a trie composed by a single node, corresponding with the zero-length pre- fix \*. Depending on the packet flow, the trie is then built to focus on the heavy prefixes. Although the re- finement process, as explained in Section [4.1,](#_bookmark11) does not depend on the selected active timeout, the node eval- uation[8](#_bookmark21), with the potential reporting to the controller,

does. This means that in the worst case scenario a full

IP address can be reported after 32 *tA* seconds: the upper bound for building the tree from the root to the lowest level. To mitigate this, we propose a variable active timeout mechanism which sets different timeout intervals and corresponding thresholds for nodes of dif- ferent prefix length, i.e., smaller timeout and threshold for shorter prefixes and vice versa. In the P4 imple- mentation we can use separate configuration registers of active timeout and threshold for each level of the tree depending on the prefix length.

*×*

# Evaluation

Following a common practice adopted for the evaluation of programmable dataplane solutions [[37,](#_bookmark66) [48],](#_bookmark77) we imple- mented a C++ simulation model of the Elastic Trie al- gorithm to assess our approach against real traffic traces from an ISP backbone and a datacenter network. Ad- ditionally, using the behavioral model available for P4 switches [[17],](#_bookmark46) we also verified the correctness of our P4 prototype by comparing its results against the outputs generated by the C++ simulation model.

In this section, we first describe our setup and we evaluate the trade-offs of the Elastic Trie data structure. Then, we discuss its detection accuracy against the sup- ported network events (hierarchical heavy hitters, su- perspreaders and traffic changes) when varying memory occupancy, type of the input traffic and data structure configuration parameters. Finally, we conclude by com- paring it with prior related solutions.

**Traces.** For the ISP backbone test case, we used four one-hour packet traces from CAIDA [[10,](#_bookmark39) [11]](#_bookmark40) recorded from 10 Gbps links in San Jose and Chicago in 2009 and 2016, respectively. All CAIDA traces are distributed in one minute chunks, and each chunk contains on average 30M packets with around 840K unique IP addresses. As for datacenter network test case, we used the publicly available traces from 2009 [[5].](#_bookmark34) These are 65 and 160 min- utes long and contain about 20M and 100M packets, re- spectively, both with around 5.5K unique IP addresses. Unfortunately, we could not use the newer datacenter

vary around zero. Otherwise, if the value of the counter

is above or below a specified threshold, it denotes a sig- 8 The process of deciding if a specific prefix is a HH or HHH.

traces from the Facebook Network Analytics Data Shar- ing program [[1],](#_bookmark30) as they were collected using sampling, which makes them inappropriate for the type of tests needed in this paper.

**Setup.** Following common practices from past re- search efforts [[37](#_bookmark66), [48]](#_bookmark77) we set the fixed active timeout *tA* to 20 seconds (measurement reporting time) and the in- active timeout *tI* to 5 minutes. The threshold *T* , used to discriminate the prefixes that are ”large enough”, has been set to be 1%, 5% and 10% of the maximum amount of traffic (packets or flows). As for the variable active timeout behavior (discussed in Section [4.3),](#_bookmark20) when adopted, we set it differently for each trie level. The set *fy*(*x*) of functions [(1)](#_bookmark22) specifies the value of the timeout for each of the trie level *x*. The coefficient *y* indicates the number of levels not being affected. For example, a value of 16 for *y* means that the first half of the trie is built using variable active timeout and the second one with a fixed timeout.

. *y tA,* if *y <* 1;

*fy*(*x*) =

32*−x*

32*−x*

(1)

threshold. The threshold *T* was set to 1%, 5% and 10% of the amount of traffic in terms of packets. The depth and number of nodes are as expected proportional to the selected threshold: the lower the threshold, the larger the depth and the number of nodes, since more prefixes are detected as heavy. It is also possible to see the learn- ing phase of the trie at the beginning of the trace, when the trie has to build up from the less specified prefix. After this phase, the trie reaches a steady state that re- flects the current traffic behavior. Figures [9b](#_bookmark25) offer, for the threshold of 5%, a more detailed view on the learn- ing phase and compare the impact of variable active timeout for different functions. Using a variable active timeout mechanism, we can speed up the learning phase by 93%, going from 300 seconds needed for fixed timeout to 20 seconds needed using the most aggressive function *f*1(*x*). As a counterpart, very aggressive functions are much more sensitive to traffic patterns, resulting in po- tential fluctuations of the trie. The last Figures [9c](#_bookmark25) show the behavior of Elastic Trie in a datacenter environment. In contrast with ISP traces, datacenter traces are much

more bursty, directly influencing the behavior of the trie.

*tA,* otherwise

This set of functions allows to have smaller timeouts for shorter prefixes, thus enabling a fine-grained control over the reporting time and the trie building capabilities. The shorter the timeouts, the smaller the amount of traffic needed to start the process of trie building, since the threshold *T* is fixed.

**Metrics.** To better understand the dynamics of the proposed data structure, we evaluated the number of nodes and the trie depth varying a number of con- figuration parameters. Then, to estimate its network event detection capabilities, we used two common met- rics [[19,](#_bookmark48) [29]:](#_bookmark58) recall and precision. Recall [(2)](#_bookmark23) is defined as the number of real events reported over the total ground truth events happened. In contrast, precision [(3)](#_bookmark24) repre- sents the total ground truth events happened over the total reported. Specifically, the recall and precision are the complements to the number of false negatives *fN* and false positives *fP* : the higher the recall the smaller the false negatives rate, while the higher the precision the smaller the false positives rate.

*real-events-reported ground-truth-events-happened*

*recall* = = 1 *− f* (2)*N*

*precision* = *ground-truth-events-happened* = 1 *− fP* (3) Unless otherwise stated, the recall and precision are al- ways indicated as the average over the chunks of the traces.

*events-reported*

## Data structure properties

Figures [9a](#_bookmark25) show Elastic Trie average depth and average number of nodes over time for CAIDA traces varying the

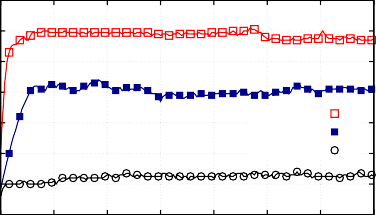
## HHH detection

In this section we first present the theoretical HHH de- tection capabilities, then our implementation driven re- sults. The former case does not take into account the im- pact of implementation details such as amount of avail- able memory or potential hash collisions. This allows us to get an understanding about the behavior of our solution in the best case scenario. The latter takes into account limitations in memory availability, as well as potential hash collisions that might happen during the classification stage. This allows us to get an understand- ing of the trade-offs between memory and detection re- sults.

**Theoretical results.** Figure [10a](#_bookmark26) and [10b](#_bookmark26) show the HHH detection capabilities in a ISP and a datacenter scenario, respectively. We used a threshold of 5%.

Since the basic behavior of Elastic Trie is to build a trie that focuses on the prefixes that account for a large share of the traffic, sometimes it might happen that the system is not quick enough in finalizing the building process when the prefix needs to be reported. Due to this, we define two ways of comparing the prefixes de- tected: exact prefix comparison and relaxed comparison, where we accept as a valid result a 2 bit coarser grained version of the prefix. The figures show that the accu- racy with exact prefix detection is lower than its 2 bit coarser grained prefix version. Overall, both recall and precision are always between 90% and 100%. The effect of variable active timeout can be seen in Figure [10a.](#_bookmark26) When using a more aggressive variable timeout the re-

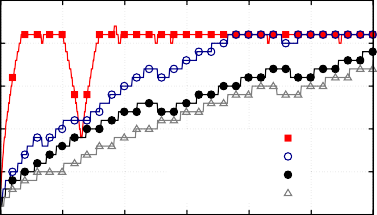
35 25 35



CAIDA 1%

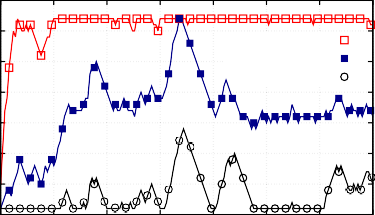
CAIDA 5%

CAIDA 10%



CAIDA - f1(x) CAIDA - f8(x) CAIDA - f16(x)

CAIDA - fxed



Datacenter 1%

Datacenter 5%

Datacenter 10%

30 30

20

25 25

Average trie depth

20 15 20

Trie depth

Trie depth

15 10 15

10 10

5

5 5

0

0 500 1000 1500 2000 2500 3000 3500

Time (seconds)

0

0 50 100 150 200 250 300

Time (seconds)

0

0 500 1000 1500 2000 2500 3000 3500

Time (seconds)

140

120

Average number of nodes

100

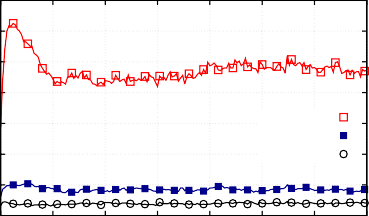
80

60

40

20

0



CAIDA 1%

CAIDA 5%

AIDA 10%

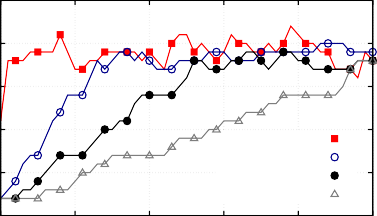
C

0 500 1000 1500 2000 2500 3000 3500

Time (seconds)

25

20



CAIDA - f1(x) CAIDA - f8(x) CAIDA - f16(x)

CAIDA - fxed

15

Nodes

10

5

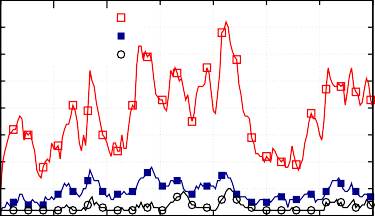
0

0 5 10 15 20 25

Time (seconds)

80

70



Datacenter 1%

Datacenter 5%

Datacenter 10%

60

50

Nodes

40

30

20

10

0

0 500 1000

1500 2000 2500 3000 3500

Time (seconds)

1. **Fixed timeout** *tA*

**in a ISP scenario.**

1. **Variable timeout** *tA*

**in a ISP scenario.**

1. **Fixed timeout** *tA*

**in datacenter scenario.**

**Figure 9: Trie depth and number of nodes varying threshold, timeout behavior and type of traffic.**

100

80

Percentage (%)

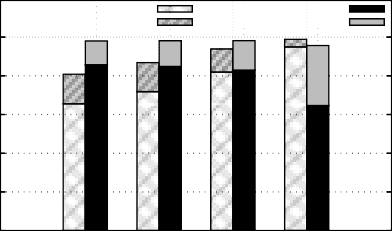
60

40

20

0

Fixed f16(x)



Recall (exact) Recall (2 bits)

Precision (exact) Precision (2 bits)

f8(x) f1(x)

sults are less accurate. This is caused by the bursty nature of datacenter traffic, which means it is more dif- ficult for the trie to build up in time. It is then clear that our solution is more suitable for an ISP scenario.

**Implementation driven results.** We assess the im- pact of the amount of available memory over the recall and precision. We find that our solution can successfully detect, with approximately 65% recall and 85% preci-

100

80

Percentage (%)

60

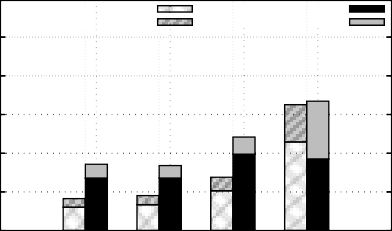
40

20

0

Window behavior

1. **ISP scenario**



Recall (exact) Recall (2 bits)

Precision (exact) Precision (2 bits)

Fixed f16(x) f8(x) f1(x)

Window behavior

1. **Datacenter scenario**

sion, the exact HHH prefix using a fixed active timeout and less than 20KB (Figure [11a).](#_bookmark27) If a coarser grained prefix is used, which is less precise by only two bits, then the recall jumps to 80% and the precision to 98%. Again, this is the consequence of the nature of the data structure: it might happen that it does not have enough time to build properly.

Using a variable timeout (Figure [11b)](#_bookmark27) results improve sensibly. In this case, it is possible to detect the exact HHH prefix with 85% recall and 90% precision with less than 8KB. Moreover, if a 2 bit coarser grained HHH pre- fix is used, the recall jumps to 95% and the precision to

**Figure 10: HHH detection capabilities varying active timeout behavior and type of traffic.**

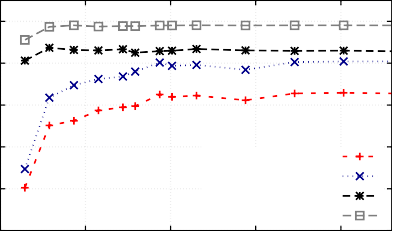
call increases, leading to a smaller false negative rate. In contrast, the precision decreases causing higher false positives rate. This is a direct effect of smaller active timeouts that lead the system to detect more prefixes. Using different functions for variable active timeout, it is then possible to fine tune the trade-off between recall and precision.

In a datacenter scenario, as shown in Figures [10b,](#_bookmark26) re-

98%. Increasing the available memory does not signifi- cantly improve the detection capabilities of the system, because it is theoretically bounded by the ability of the trie to react and build up according to the input traffic patterns.

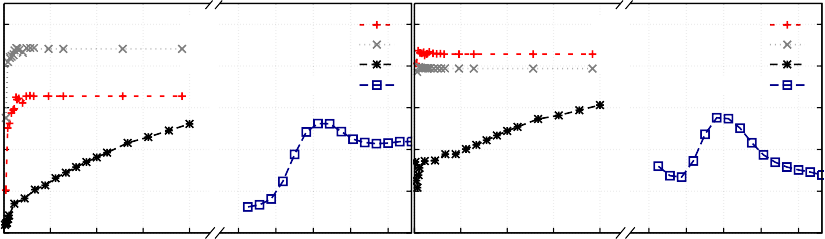
In Figure [12,](#_bookmark27) we compare the HHH detection capabil- ities of Elastic Trie against related prior programmable dataplane based solutions: UnivMon [[37]](#_bookmark66) and Hash- Pipe [[48].](#_bookmark77) UnivMon and HashPipe use an alternative definition for HH detection, named the “top-*k* problem”. Instead of reporting prefixes that are larger than a given threshold, they report the top-*k* sources, no matter the

100 100



Recall (exact)

Recall (2 bits) Precision (exact) Precision (2 bits)



E

ic Trie (fxed)

Trie (variable) HashPipe

UnivMon

E

ic Trie (fxed)

Trie (variable) HashPipe

UnivMon

Elast

lastic

Elast

lastic

80 80

Percentage (%)

Percentage (%)

60 60

40 40

20 20

0

0 5 10 15 20

0

0 36 72 108 144

600 700 800 900 1000 0 36 72 108 144

600 700 800 900 1000

Memory (KB)

* 1. **Fixed timeout** *tA*

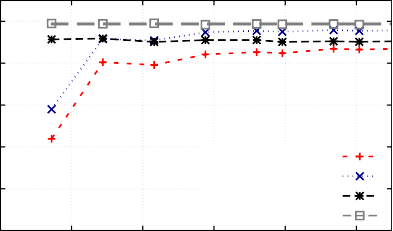
Memory (KB)

1. **Recall (exact)**

Memory (KB)

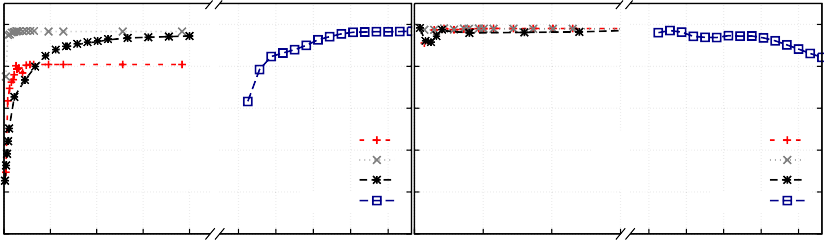
1. **Precision (exact)**

100 100



Recall (exact)

Recall (2 bits) Precision (exact) Precision (2 bits)



E

riable)

shPipe UnivMon

E

riable)

shPipe UnivMon

rie (va Ha

lastic T

(fxed)

c Trie

Elasti

Trie (va Ha

lastic

(fxed)

ic Trie

Elast

80 80

Percentage (%)

Percentage (%)

60 60

40 40

20 20

0

0 2 4 6 8 10

0

0 36 72 108 144

600 700 800 900 1000 0 10 20 30 600 700 800 900 1000

Memory (KB)

1. **Variable timeout** *tA*

Memory (KB)

1. **Recall (2 bits)**

Memory (KB)

1. **Precision (2 bits)**

**Figure 11: HHH detection capa- bilities varying memory occupancy and active timeout behavior.**

**Figure 12: Comparison between Elastic Trie, UnivMon, and Hashpipe of Hierarchical Heavy Hitter detection capabilities in ISP scenario.**

amount of traffic they are actually sending. To per- form a fair comparison, and align their results with the one produced by our system (which follows the classic HHH definition), we aggregated their output addresses into prefixes and considered only the ones that carry traffic beyond the fixed threshold *T* . Figures [12a](#_bookmark27) and [12b](#_bookmark27) show the results using an exact prefix compari- son. HashPipe needs a much lower amount of mem- ory ( 144KB) than UnivMon ( 800KB) to reach recall and precision around 50-60%. In contrast, Elastic Trie significantly outperforms both of the solutions. The in- accuracies spotted in UnivMon and HashPipe can be partially related to the different definition of heavy hit- ters being used. This is also confirmed by the results obtained when a coarser grained prefix is used (Fig- ures [12c](#_bookmark27) and [12d](#_bookmark27)). Nevertheless, the memory require- ments of the three solutions represent a fair comparison metric. HashPipe and Elastic Trie have the same mem- ory requirements, but HashPipe can only detect Heavy Hitters, while our solution adds more network events. UniMon is not restricted to a single network event, but requires 90% more memory to work.

*∼ ∼*

## Superspreader detection

As in the HHH case, we first introduce the theoretical results (without taking into account available memory or hash collisions). Then, we show the trade-offs be- tween memory occupancy and superspreader detection

capabilities.

**Theoretical results.** Figure [13a](#_bookmark28) shows the theoret- ical superspreader detection recall and precision capa- bilities for CAIDA traces varying the active timeout be- havior. In contrast to the same evaluation for HHH de- tection, Elastic Trie superspreader detection, using vari- able timeout, is less good at detecting the exact prefix length than when using a fixed timeout. In this case, it is clearer that the trie cannot build in time, as both recall and precision grow sensibly when we use a 2 bit coarser grained superspreader prefix. Overall, for fixed active timeout the detection capabilities are still good, as both recall and precision are around 80% and 95%.

**Implementation driven results.** In Figure [13b](#_bookmark28) we show the impact of available memory over the detection capabilities, taking into account our P4 implementation. For this test, we used a fixed active timeout, fixed 25KB of memory for the allocation of the prefix trie struc- ture, and we varied the Bloom filter occupancy. We find that superspreaders can be successfully detected with approximately 60% recall and 85% precision and 78% recall and 98% precision when a 2 bit coarser grained prefix is used, respectively, with less than 250KB of al- located memory.

## Change detection

To demonstrate traffic change detection capabilities of Elastic Trie structure, we artificially injected network

100

80

Percentage (%)

60

40

20

0

Fixed f16(x)



Precision (exact) Precision (2 bits)

ll (exact) ll (2 bits)

Reca Reca

f8(x) f1(x)

# Related Work

SDN-based monitoring solutions which rely on statistics retrieval from switches [[50,](#_bookmark79) [54,](#_bookmark83) [15]](#_bookmark44) might suffer from lim- ited visibility. As demonstrated in Section [2,](#_bookmark3) this can be a very expensive process, that can overload either the controller or the switch itself. In contrast, our solution reports to the controller the network events of interest

100

80

Percentage (%)

60

40

20

0

Window behavior

* + 1. **Variable timeout** *tA*

Recall (exact)

Recall (2 bits) Precision (exact) Precision (2 bits)

0 50 100 150 200 250

Memory (KB)

* + 1. **Memory occupancy**

as soon as they happen, without the need of a central

controller. To have flexibility in identifying the interest- ing flows, iterative refinement of monitored flows can be used, which can be costly for the control channel, since it requires flow updates to zoom in the traffic of inter- est [[29,](#_bookmark58) [57,](#_bookmark86) [41].](#_bookmark70) While our architecture also relies on iterative refinement when building the trie to focus on the flows of interest, it does so in the data plane, with no direct intervention from the control plane. Algorithms that use iterative refinement of flows to determine heavy hitters and anomalies were presented in [[56]](#_bookmark85) and [[31],](#_bookmark60) re- spectively, but they were targeted for custom measure-

**Figure 13: Superspreader detection capabilities.**

traffic simulating DoS attack and scanning into one of the CAIDA traces. The attack has been emulated after 2500 seconds since the beginning of the trace. DoS and scanning are two type of attacks that can potentially change traffic patterns. At the same time, they are also pretty different: while a DoS is typically a source that sends a huge amount of traffic to a designated victim, the scan is a source contacting many random destinations. Figure [14a](#_bookmark29) shows the time on the x-axis and a moving average of trie changes (difference between number of expanded and collapsed nodes) on the y-axis. Note that tree is built based on HHH detection using a fixed active timeout *tA* = 20 seconds. In the figure we can distinctly see differences during normal conditions and the state under DoS attack or scan. After a learning phase, our data structure can be used to verify if the input traffic patterns suddenly change. Figure [14b](#_bookmark29) shows the same situation but from a different perspective. Now the trie is built on top of the superspreader detection. In this case, the DoS attack is not detected at all, because it represents a communication with only one distinct des- tination. On the other hand, the scan, as a typical case of superpreader, is much more significant now. The last Figure [14c](#_bookmark29) shows the situation based again on HHH de- tection, but using the variable active timeout mecha- nism. Due to accelerated trie construction, there are many more changes in the trie over a short time period. This allows to highlight further even small changes in the traffic patterns, as shown when comparing the scan behavior for Figure [14a](#_bookmark29) and Figure [14c.](#_bookmark29)

ment platforms (not match-action type architecture).

More recently, a number of monitoring frameworks leveraging P4 programmability have been developed [[36,](#_bookmark65) [37,](#_bookmark66) [48,](#_bookmark77) [43].](#_bookmark72) FlowRadar [[36]](#_bookmark65) keeps track of all the flows in the network and their counters, and exports this in- formation periodically to a remote collector, which de- codes it and uses it for various monitoring applications targeted to datacenters. Our aim is to not keep track of all the flows in the network as FlowRadar does, but to be able to efficiently detect network events related to high-volume traffic clusters from within the dataplane. UnivMon [[37]](#_bookmark66) uses a general sketch in the dataplane to keep track of the flows, which offers information for sev- eral monitoring applications, and is exported at fixed time intervals to the control plane. HashPipe [[48]](#_bookmark77) deter- mines the top-k heavy hitters in the dataplane, and ex- ports them at fixed time intervals. Our work informs the controller as soon as the considered network events take place, without having to wait for the end of the time in- terval. Sonata [[26],](#_bookmark55) which proposes a query interface for network telemetry, uses sketches in the dataplane, and zooms-out the network traffic of interest by refining the network query, starting from the finest level. The query refinement is done by the controller, who programs the new query plan on the target in each iteration, while in our case, the refinement is done directly in the dat- aplane. While [[43]](#_bookmark72) presents a solution for determining hierarchical heavy hitters directly from the dataplane, their solution does not cover other measurements tasks.

# Conclusion

In this paper, we proposed a push-based approach to network monitoring, where the dataplane informs the

7

DoS attack

Scan attack

Normal condition

Average number of trie node changes

6

5

4

3

2

1

0

-1

-2

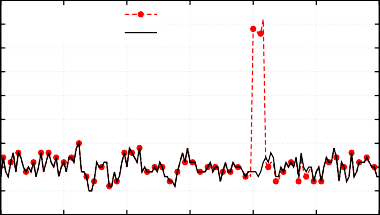
500 1000 1500

2000 2500 3000 3500

Time (seconds)

7

6



Scan attack

Normal condition

Average number of trie node changes

5

4

3

2

1

0

-1

-2

500 1000 1500 2000 2500 3000 3500

Time (seconds)

7

6



Average number of trie node changes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DoS attack Scan attack Normal condition | |  |  |  |  |  |
|  |  |  |  |
|  | |  |  |  |
|  |  |  | |  |  |  |
|  |  |  | |  |  |  |
|  |  |  | |  |  |  |
|  |  |  | |  |  |  |
|  |  |  | |  |  |  |
|  |  |  | |  |  |  |

5

4

3

2

1

0

-1

-2

500 1000 1500

2000 2500 3000 3500

Time (seconds)

1. **Fixed active timeout, using (H)HH detection.**
2. **Fixed active timeout, using superspreaders.**
3. **Variable active timeout, using (H)HH detection.**

**Figure 14: Change detection capabilities varying active timeout and trie building behavior.**

control plane only when specific conditions are met. To achieve this, we presented a new data structure, Elas- tic Trie, that enables the detection of a number of net- work events associated with high-volume traffic clusters within the dataplane. Our solution has been designed with the constraints of emerging programmable switches in mind, as it works in a packet-driven manner, and can be implemented using common match-action based ar- chitectures such as RMT.

Elastic Trie uses a hash table based prefix tree that grows or collapses to focus only on the prefixes that account for a ”large enough” share of the traffic. This enables the detection of (hierarchical) heavy hitters, and by looking at its growing rate it is possible to identify changes in the traffic patterns. We prototype our solu- tion with P4 and demonstrate its detection capabilities. Specifically, using simulation on real traffic traces taken from an ISP backbone and a datacenter, we showed that Elastic Trie achieves high accuracy in detecting hierar- chical heavy hitters, superpreaders and changes in the network traffic patterns with the memory constraints imposed by today’s switches.

# References

1. Data Sharing on traffic pattern inside Facebook’s datacenter network, Jan 2018. [https://research.fb.com/data-sharing-](https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/)

[on-traffic-pattern-inside-facebooks-](https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/) [datacenter-network/](https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/).

1. sFlow, Jan 2018. [http://www.sflow.org/about/ index.php](http://www.sflow.org/about/index.php).
2. M. Al-Fares, S. Radhakrishnan, B. Raghavan,

N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2010.

1. R. Ben Basat, G. Einziger, and R. Friedman. Fast Flow Volume Estimation. In *Proceedings of the 19th International Conference on Distributed Comput- ing and Networking*, ICDCN ’18, pages 44:1–44:10, New York, NY, USA, 2018. ACM.
2. T. Benson. Data Set for IMC 2010 Data Center Measurement, Jan 2018. [http://pages.cs.wisc. edu/~tbenson/IMC10\_Data.html](http://pages.cs.wisc.edu/%7Etbenson/IMC10_Data.html).
3. T. Benson, A. Anand, A. Akella, and M. Zhang. Mi- croTE: Fine Grained Traffic Engineering for Data Centers. In *COnference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2011.
4. B. H. Bloom. Space/Time Trade-offs in Hash Cod- ing with Allowable Errors. In *Communications of the ACM (CACM), Volume: 13, Issue: 7*. ACM, 1970.
5. P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McK- eown, J. Rexford, C. Schlesinger, D. Talayco,
   1. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Proces- sors. In *SIGCOMM Computer Communication Re- view, Volume: 44, Issue: 3*. ACM, 2014.
6. P. Bosshart, G. Gibb, H.-S. Kim, G. Vargh- ese, N. McKeown, M. Izzard, F. Mujica, and

M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hard- ware for SDN. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.

1. CAIDA. Anonymized Internet Traces 2009 – 17th September 2009, Jan 2018. [http://www.caida. org/data/passive/passive\_2009\_dataset.xml](http://www.caida.org/data/passive/passive_2009_dataset.xml).
2. CAIDA. Anonymized Internet Traces 2016 – 17th March 2016, Jan 2018. [http://www.caida.org/ data/passive/passive\_2016\_dataset.xml](http://www.caida.org/data/passive/passive_2016_dataset.xml).
3. C. Callegari, S. Giordano, M. Pagano, and T. Pepe. Detecting Anomalies in Backbone Network Traffic: A Performance Comparison Among Several Change Detection Methods. Inderscience Publishers, 2012.
4. K. Cho. Recursive Lattice Search: Hierarchical Heavy Hitters Revisited. In *Proceedings of the 2017 Internet Measurement Conference*, IMC ’17, pages 283–289, New York, NY, USA, 2017. ACM.
5. B.-Y. Choi, J. Park, and Z.-l. Zhang. Adaptive ran- dom sampling for traffic load measurement. In *In- ternational Conference on Communications (ICC)*. IEEE, 2003.
6. S. R. Chowdhury, M. F. Bari, R. Ahmed, and

R. Boutaba. PayLess: A low cost network monitor- ing framework for Software Defined Networks. In *Network Operations and Management Symposium (NOMS)*. IFIP/IEEE, 2014.

1. B. Claise. Cisco Systems NetFlow Services Export Version 9, Jan 2018. [https://tools.ietf.org/ html/rfc3954](https://tools.ietf.org/html/rfc3954).
2. P. L. Consortium. P4 Switch Behavioral Model, Jan 2018. [https://github.com/p4lang/behavioral- model](https://github.com/p4lang/behavioral-model).
3. T. P. L. Consortium. *P* 416 Language Specification, version 1.0.0, Jan 2018. [http://p4.org/p4-spec/ docs/P4-16-v1.0.0-spec.pdf](http://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf).
4. G. Cormode, F. Korn, S. Muthukrishnan, and

D. Srivastava. Finding Hierarchical Heavy Hitters in Streaming Data. In *Transactions on Knowledge Discovery from Data, Volume: 1, Issue: 4*. ACM, 2008.

1. A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagan- dula, P. Sharma, and S. Banerjee. DevoFlow: Scal- ing Flow Management for High-performance Net- works. In *Special Interest Group on Data Commu- nication (SIGCOMM)*. ACM, 2011.
2. N. Duffield, C. Lund, and M. Thorup. Charging from Sampled Network Usage. In *Internet Mea- surement Workshop (IMW)*. ACM, 2001.
3. C. Estan, K. Keys, D. Moore, and G. Vargh- ese. Building a Better NetFlow. In *Special Inter- est Group on Data Communication (SIGCOMM)*. ACM, 2004.
4. C. Estan and G. Varghese. New Directions in Traf- fic Measurement and Accounting. In *Special Inter- est Group on Data Communication (SIGCOMM)*. ACM, 2002.
5. A. Feldmann, A. Greenberg, C. Lund, N. Reingold,

J. Rexford, and F. True. Deriving Traffic Demands for Operational IP Networks: Methodology and Ex- perience. In *Transactions on Networking, Volume: 9, Issue: 3*. IEEE/ACM, 2001.

1. V. Giotsas, P. Richter, G. Smaragdakis, A. Feld- mann, C. Dietzel, and A. Berger. Inferring BGP Blackholing Activity in the Internet. In *Internet Measurement Conference (IMC)*. ACM, 2017.
2. A. Gupta, R. Harrison, A. Pawar, R. Birkner,

M. Canini, N. Feamster, J. Rexford, and W. Will- inger. Sonata: Query-Driven Network Teleme- try, Jan 2018. [https://arxiv.org/abs/1705. 01049v1](https://arxiv.org/abs/1705.01049v1).

1. K. He, J. Khalid, A. Gember-Jacobson, S. Das,

C. Prakash, A. Akella, L. E. Li, and M. Thottan. Measuring Control Plane Latency in SDN-enabled Switches. In *Symposium on Software Defined Net- working Research (SOSR)*. ACM, 2015.

1. Q. He, Y. Wang, W. Li, and X. Qiu. Traffic steer- ing of middlebox policy chain based on SDN. In *Integrated Network and Service Management (IM)*. IFIP/IEEE, 2017.
2. L. Jose, M. Yu, and J. Rexford. Online Measure- ment of Large Traffic Aggregates on Commodity Switches. In *Hot Topics in Management of Inter- net, Cloud, and Enterprise Networks and Services (Hot-ICE)*. USENIX, 2011.
3. J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash Crowds and Denial of Service Attacks: Char- acterization and Implications for CDNs and Web Sites. In *World Wide Web (WWW)*. ACM, 2002.
4. F. Khan, N. Hosein, C.-N. Chuah, and S. Ghi- asi. Streaming Solutions for Fine-Grained Network Traffic Measurements and Analysis. In *Architec- tures for Networking and Communications Systems (ANCS)*. IEEE Computer Society, 2011.
5. B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based Change Detection: Methods, Evalu- ation, and Applications. In *Internet Measurement Conference (IMC)*. ACM, 2003.
6. A. Lakhina, M. Crovella, and C. Diot. Diagnos- ing Network-wide Traffic Anomalies. In *Computer Communication Review, Volume: 34, Issue: 4*. ACM, 2004.
7. A. Lakhina, M. Crovella, and C. Diot. Mining Anomalies Using Traffic Feature Distributions. In *Computer Communication Review, Volume: 35, Is- sue: 4*. ACM, 2005.
8. I. Lenovo. Rackswitch g8264 product guide, Jan 2018. <https://lenovopress.com/tips0815>.
9. Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2016.
10. Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and

V. Braverman. One Sketch to Rule Them All: Re- thinking Network Flow Monitoring with UnivMon. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

1. J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and

H. Zang. Is Sampled Data Sufficient for Anomaly Detection? In *Internet Measurement Conference (IMC)*. ACM, 2006.

1. N. McKeown, T. Anderson, H. Balakrishnan,

G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. In *SIGCOMM Computer Com- munication Review, Volume: 38, Issue: 2*. ACM, 2008.

1. M. Mitzenmacher, T. Steinke, and J. Thaler. Hier- archical Heavy Hitters with the Space Saving Algo- rithm. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, ALENEX ’12, pages 160–174, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.
2. M. Moshref, M. Yu, R. Govindan, and A. Vah- dat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Special Inter- est Group on Data Communication (SIGCOMM)*. ACM, 2014.
3. Noviflow. Noviswitch 1132 product guide, Jan 2018. [https://noviflow.com/wp-content/uploads/ NoviSwitch-1132-Datasheet-V2.0.pdf](https://noviflow.com/wp-content/uploads/NoviSwitch-1132-Datasheet-V2.0.pdf).
4. D. A. Popescu, G. Antichi, and A. W. Moore. En- abling Fast Hierarchical Heavy Hitter Detection Us- ing Programmable Data Planes. In *Symposium on SDN Research (SOSR)*. ACM, 2017.
5. Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy En- forcement Using SDN. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.
6. J. Rasley, B. Stephens, C. Dixon, E. Rozner,

W. Felter, K. Agarwal, J. Carter, and R. Fon- seca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *Special Inter- est Group on Data Communication (SIGCOMM)*. ACM, 2014.

1. J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy,

S. Ratnasamy, and V. Sekar. Making Middleboxes Someone else’s Problem: Network Processing As a Cloud Service. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2012.

1. A. Sivaraman, S. Subramanian, M. Alizadeh,

S. Chole, S.-T. Chuang, A. Agrawal, H. Balakr- ishnan, T. Edsall, S. Katti, and N. McKeown. Pro- grammable Packet Scheduling at Line Rate. In *Spe- cial Interest Group on Data Communication (SIG- COMM)*. ACM, 2016.

1. V. Sivaraman, S. Narayana, O. Rottenstreich,

S. Muthukrishnan, and J. Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *Sympo- sium on SDN Research (SOSR)*. ACM, 2017.

1. V. Srinivasan and G. Varghese. Faster IP Lookups Using Controlled Prefix Expansion. In *Special In- terest Group on performance evaluation (SIGMET- RICS)*. ACM, 1998.
2. A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: Traffic Matrix Estimator for OpenFlow Networks. In *Passive and Active Measurement (PAM)*. Springer-Verlag, 2010.
3. S. Venkataraman, D. Song, P. B. Gibbons, and

A. Blum. New Streaming Algorithms for Fast De- tection of Superspreaders. In *Network and Dis- tributed System Security Symposium (NDSS)*. In- ternet Society, 2005.

1. S. Venkataraman, D. Song, P. B. Gibbons, and

A. Blum. New Streaming Algorithms for Fast De- tection of Superspreaders. In *Network and Dis- tributed System Security Symposium (NDSS)*. In- ternet Society, 2005.

1. Y. Xie, V. Sekar, D. A. Maltz, M. K. Reiter, and

H. Zhang. Worm Origin Identification Using Ran- dom Moonwalks. In *Security and Privacy (SP)*. IEEE Computer Society, 2005.

1. C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha. FlowSense: Monitor- ing Network Utilization with Zero Measurement Cost. In *Passive and Active Measurement (PAM)*. Springer-Verlag, 2013.
2. M. Yu, L. Jose, and R. Miao. Software De- fined Traffic Measurement with OpenSketch. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2013.
3. L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards Programmable Network Measurement. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2007.
4. Y. Zhang. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *Conference on Emerging Networking Experiments and Technolo- gies (CoNEXT)*. ACM, 2013.
5. Y. Zhang, S. Singh, S. Sen, N. Duffield, and

C. Lund. Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Ap- plications. In *Internet Measurement Conference (IMC)*. ACM, 2004.