Enabling Event-Triggered Data Plane Monitoring

Jan Kučera
CESNET
jan.kucera@cesnet.cz

Diana Andreea Popescu
University of Cambridge
diana.popescu@cl.cam.ac.uk

Han Wang
Barefoot Networks
hanwang@barefootnetworks.com

Andrew Moore
University of Cambridge
andrew.moore@cl.cam.ac.uk

Jan Kořenek
Brno University of Technology
Centre of Excellence IT4Innovations

Gianni Antichi
Queen Mary University of London
g.antichi@qmul.ac.uk

ABSTRACT

We propose a push-based approach to network monitoring that allows the detection, within the dataplane, of traffic aggregates. Notifications from the switch to the controller are sent only if required, avoiding the transmission or processing of unnecessary data. Furthermore, the dataplane iteratively refines the responsible IP prefixes, allowing the controller to receive information with a flexible granularity. We implemented our solution, Elastic Trie, in P4 and for two different FPGA devices. We evaluated it with packet traces from an ISP backbone. Our approach can spot changes in the traffic patterns and detect (with 95% of accuracy) either hierarchical heavy hitters with less than 8KB or superspreaders with less than 300KB of memory, respectively. Additionally, it reduces controller-dataplane communication overheads by up to two orders of magnitude with respect to state-of-the-art solutions.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Network measurement**; *Programmable networks*; *In-network processing*.

KEYWORDS

Network measurements, traffic aggregates, Elastic Trie, P4.

ACM Reference Format:
Jan Kučera, Diana Andreea Popescu, Han Wang, Andrew Moore, Jan Kořenek, and Gianni Antichi. 2020. Enabling Event-Triggered Data Plane Monitoring. In *Symposium on SDN Research (SOSR '20), March 3, 2020, San Jose, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3373360.3380830

1 INTRODUCTION

Network management practices can be performed efficiently if high-volume traffic clusters are promptly detected [20, 24, 32, 39, 48]. Indeed, spotting a single source or destination that sends or receive a significant amount of data (heavy hitter) is beneficial for accounting [21, 24], or traffic engineering [6, 25]. In contrast, detecting a source that reaches multiple distinct destinations (superspreader) is needed for worm or scan detection [50, 51]. Finally, finding which flow contributes the most to the traffic pattern changes in a short period of time (change detection) is of paramount importance in the context of anomaly detection [36, 37]. All of the aforementioned events exhibit high-volume traffic clusters in a different way: while in the context of heavy hitters the "cluster" relates to packets or bytes arrival rate, for superspreaders the interest shifts to the flows arrival rate.

In the past, the detection of those events were performed outside the dataplane in software collectors. Switches, to lower overheads and data collection bandwidth at the cost of estimation accuracy [14, 22, 40], employed packet sampling and exported statistics using well known protocols such as NetFlow [16] or sFlow [2]. Lately, the advent of programmable switches [8] has enabled the possibility of extending dataplane functionality with more advanced traffic analysis features. Nonetheless, current devices have constrained resources, requiring clever solutions to deal with both computation and memory limitations. Such restrictions have led the research community to deal with a specific trade-off: while a single use-case can be easily enabled in the dataplane, scaling to more requires the help of a central controller. Indeed, recent proposals that push more computation in the switch focus only on one specific goal: detection of the top-k heavy hitters [48] or microbursts [13]. In contrast, solutions that aim for a more generic approach aggregate traffic information in probabilistic data structures, i.e, sketches [30, 39, 52, 54], which are then entirely exported to a controller for analysis. Despite the use of sketches results in a very flexible and generic approach to network monitoring, the controller still needs to receive the generated information from the dataplane at a fixed time interval, and then estimate the various application-level metrics of interest. Such an architecture has similar drawbacks to that of the OpenFlow (OF) protocol: 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 scales [20].

In this paper we start from the observation that important network management practices [20, 24, 32, 39, 48], i.e., traffic engineering, security, benefit if heavy hitters, superspreaders and traffic pattern changes are promptly detected. We thus build a solution which is capable of detecting the mentioned network events entirely in the dataplane, by iteratively tracking the responsible IP prefixes and only subsequently informing the controller. We designed a new data structure, *Elastic Trie*, with the constraints of emerging programmable switches in mind, and present its implementation in both P4 and for two different FPGA devices. The idea is to build in the switch a prefix tree that continuously grows or collapses to focus only on the prefixes that account for a "large enough" share of the traffic.

SOSR '20, March 3, 2020, San Jose, CA, USA

ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380830

Network event	Management task
(Hierarchical) Heavy Hitters	accounting [21, 24], traffic engineering [6, 25]
Changes in traffic patterns	anomaly detection [36, 37]
Superspreaders	worm [51], scan [50], DDoS detection [54]

Table 1: Three network events for many use-cases.

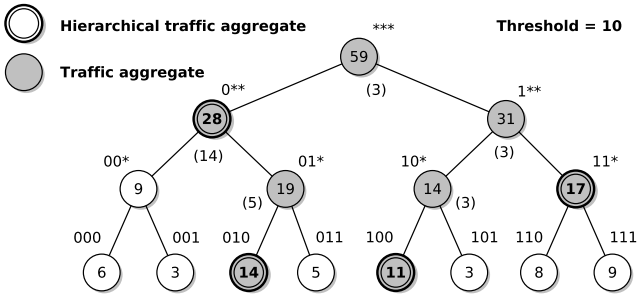


Figure 1: Example of pure (gray nodes) and hierarchical (double circle nodes) traffic aggregates following the generic definition. Each node represents a prefix p with associated amount of traffic.

This enables the detection of either (hierarchical) heavy hitters or superspreaders, and at the same time by looking at its growing rate it is possible to identify changes in the traffic patterns. Elastic Trie (ET) shares some high level principles with recent solutions for network monitoring such as Marple [42] and Sonata [28], but it is fundamentally different. Marple focuses mostly on flow performance metrics and not on traffic aggregates. In contrast, Sonata enables operators to get insights on traffic volumes and anomalies, but both requires a central controller to iteratively refine the query to efficiently capture only the traffic that pertains to the operator's query. ET performs such a refinement within the dataplane, thus completely offloading the control path.

The main contributions of the paper are as follows:

- We propose a *push-based* approach to network monitoring, where the dataplane informs the control plane only when a specific network event is detected.
- We present a data structure that enables the detection of changes in network traffic and, at the same time, detects either hierarchical heavy hitters or superspreaders entirely in the dataplane. Our solution iteratively refines the responsible prefixes so that the controller receives a finer or coarser grained information depending on the desired reporting time.
- We implemented our idea in P4 using match-action tables and for two different FPGA devices. We finally demonstrate its performance in terms of throughput and latency, and its detection capabilities by evaluating it through trace-driven simulations.

The remainder of the paper is organized as follows. We first provide a generic definition of high-volume traffic aggregates and related events (§2). We then concentrate on challenges in the aggregate detection motivating a new solution (§3) and discuss its desired properties (§4). We then present ET, our solution (§5), alongside the prototype implementation (§6) and the experimental evaluation (§7). Finally, we cover related works (§8) and conclude the paper (§9).

2 THREE PRIMITIVES, MANY USE-CASES

A broad spectrum of use-cases can be enabled with the detection of (hierarchical) heavy hitters, superspreaders and changes in the traffic patterns. This section provides a generic definition for traffic clusters and discusses how those events are related, while Tab. 1 links them to the specific use-case.

2.1 High-volume traffic clusters

A high-volume traffic cluster (or aggregate) can be defined as a prefix exceeding a pre-determined threshold in a time window [19]. Assuming the use of the source IP address as a key, the goal of the clusters detection problem is to find the source IP prefixes that contribute with a traffic volume, in terms of bytes, packets or flows, larger than a threshold T during a time interval t. The threshold T can be also specified as a percentage of the total number of inputs. However, in real live monitoring, it is not possible to know the number of inputs in advance, thus the threshold T has to be estimated, e.g., based on the number of inputs during the previous time interval t.

Fig. 1 depicts an example of traffic aggregate prefixes following the previous definition. Each node of the tree represents a prefix p in a reduced 3-bit model domain of IP addresses and its associated amount of traffic. Terminal nodes express only the traffic volume produced by full IP addresses. Non-terminal nodes summarize the traffic of a prefix p. The contribution of each prefix is represented as a number in each node. Considering the use of a threshold $T = 10$, nodes `010`, `100`, `11*` and all their ancestors are identified as high-volume aggregates. For example, each child of the `11*` node contributes independently less than the threshold T, but in total both children contribute enough to exceed the threshold and report the `11*` prefix as an aggregate.

A hierarchical aggregate is a special case of traffic aggregate [19]. It is a prefix p, which exceeds a threshold T after excluding the contribution of all its high-volume descendants[1]. In Fig. 1, only prefixes `010`, `100`, `0**` and `11*` are hierarchical aggregates. The amount of traffic of each aggregate prefix without the impact of its hierarchical descendants is shown in brackets. In this example, the `11*` node is a hierarchical aggregate, 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 `1**` prefix is not hierarchical because a significant part of its contribution originates from its descendants `100` and `11*`, which are already hierarchical aggregates and must be excluded. It is worth noting that, while the detection of hierarchical aggregates requires the knowledge of pure high-volume traffic clusters, the

[1]The descendant prefixes need to satisfy the definition of high-volume traffic aggregate.

opposite is not true. Reporting the hierarchical aggregates to a controller guarantees minimum overhead, while providing all the necessary information. Taking Fig. 1 as an example, a switch capable of detecting traffic aggregates would export the following prefixes: `0**`, `1**`, `01*`, `10*`, `11*`, `010` and `100`. In contrast, a switch reporting just hierarchical aggregates would provide `0**`, `11*`, `010` and `100`. In both cases the amount of useful information is the same[2], but with hierarchical aggregates we export less data.

2.2 Traffic clusters events

Given the previous definition of high-volume traffic clusters, we show how heavy hitter, change detection and superspreader network events fit into it.

a) Heavy hitter. A heavy hitter (HH) [19] is defined to be a host that sends or receives at least a given number of packets (or bytes) over a short period of time. It is a traffic cluster in terms of packets (or bytes) per second.

b) Change detection. Change detection is the practice of finding which flows contribute the most to the traffic pattern changes over two consecutive time intervals [12]. The method detects 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 [35]. It is a traffic cluster in terms of packets (or bytes) per second.

c) Superspreader. A superspreader (SS) is defined to be a host that contacts at least a given number of distinct destinations over a short time period. It is a traffic cluster in terms of unique flows per second. In addition, if the same spread detection is applied to the destination, this analysis allows Denial of Service (DDoS) victim detection [54].

3 MOTIVATING A NEW SOLUTION

State-of-the-art solutions that allow the detection of high-volume traffic clusters, periodically export aggregated flow counters to a controller which ultimately is in charge of estimating the metrics of interest [30, 39, 52, 54]. Nevertheless, such an architectural choice requires a careful controller-dataplane coordination.

The reporting time dilemma. *When shall I export my data structure to a central controller?* We ran a first experiment to estimate the importance of setting the correct reporting time. In the context of heavy flow detection, let us assume a flow is indexed only through the packet source IP and the switch has enough memory to keep track of every flow. Let us also assume that the switch exports periodically the counters, and the controller, in charge of the detection, considers heavy the flows that exceed 1% of the total traffic. Finally, let us consider a reporting time of 20 seconds, as suggested by state-of-the-art solutions [39, 48]. We analyzed four one-hour packet traces from CAIDA [9, 10] and we split them into 720 chunks of 20 seconds each. We then computed the heavy flows based on the previous definition. Finally, we decreased the reporting time and we calculated which of the heavy flows could have been detected earlier. Fig. 2 reports the CDF of reported heavy flows varying the reporting time. Interestingly, on average, more than 60% of the heavy flows could have been detected within one second. Note that

[2]Some of the reported high-volume traffic aggregates are just prefixes of more specific hierarchical traffic aggregates.

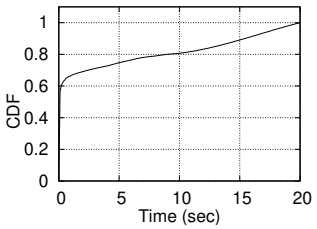


Figure 2: CDF for heavy flow detection time.

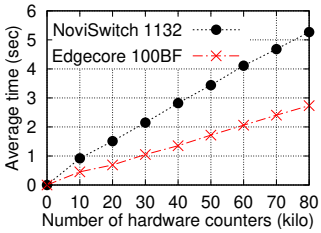


Figure 3: Time to retrieve hardware counters.

the results are based on offline analysis of packet traces only and thus do not contain false positives. This test suggests that, in theory, the reporting time should be set as low as possible. However, in practice, it is important to take into account the switch-controller capabilities of collecting statistics at short time scales, which we quantify in the next experiment.

The cost of statistics collection. *Is it just about exporting the data structure at very short timescales?* We ran an experiment to measure the amount of time it takes to retrieve an increasing number of hardware counters from a switch. We used two different hardware based systems. An OpenFlow-enabled switch, NoviSwitch 1132 [43], which has been designed for use in high bandwidth and flow-intensive network deployments, and a P4-enabled device, the Wedge 100BF-32X, a white box from Edgecore. We connected the switches to a server running a controller and we built an application that allows to request an increasing number of flow counters. The switches were idle when the counters were pulled. Fig. 3 shows the results we obtained. Although the use of probabilistic data structures, i.e., sketches, can help in reducing the number of counters to be exported, past research has shown that from around 60K to 150K counters are still required to provide useful information to a controller [39, 52]. In this context, the NoviSwitch needs at least 5 seconds, and the Edgecore 2.5 seconds. The lesson learned is that retrieving a large amount of data from hardware is time consuming and requires time scales of seconds. This finding has two major implications: (1) given that the statistics retrieval process is performed periodically, the operation needs to be dimensioned with respect to the switch capabilities: the reporting time cannot be lower than the time needed to collect the statistics; (2) in the worst case scenario, a controller can apply appropriate network updates only seconds after a specific event has happened. Therefore, performing traffic analysis in the controller might introduce delays that depending on the specific use-case are not acceptable, e.g., (D)DoS detection. On the other hand, pushing notifications to a controller as soon as an event is detected in the dataplane would allow a reaction in a more timely fashion.

The limited memory access. *Would a push-based sketch work then?* Programmable switches based on match-action architectures, i.e., RMT [8], process packets in a pipeline and for stateful processing (aggregation of flow counters) use a small amount of SRAM that persists across consecutive packets. To guarantee high throughput, the complexity of pipeline stages is limited: this impacts the number of memory accesses. Only one or a few addresses in the memory block can be read or written from a dataplane algorithm, but due

to per-stage timing constraints not the entire memory region [4, 8]. Hence, it is not possible to package the entire counter-based or sketch-based data structure in a single In-Band Network Telemetry (INT) style packet [27, 31]. Such a limitation opens up a specific question:

> *Is it possible to design a data structure, well-suited for a push-based design, that would access only a small memory block and expose a single entry upon the detection of a network event?*

4 DESIRED PROPERTIES

Fig. 4 surveys the design space for the detection of high volume traffic clusters and places our solution, *Elastic Trie*, in the context by following the thick red lines through the design tree. This section describes the insights that inform our major design decisions.

Push-based friendly. Given today's constraints of programmable switches, only a very small memory block can be read at once and sent to a controller with a single digest packet [4, 8]. Thus, to support push-based notifications, the design of the data structure is of paramount importance: it needs to quickly locate the memory address where the prefix to be announced to a controller is stored. Let us take a sketch data structure as an example: in order to find the most populated bucket and send its related information in a digest packet, a program should first scan all the possible entries. This is clearly not optimal. Indeed, current sketch-based architectures work with a poll-based mode [30, 39, 52, 54], where the controller retrieves the whole data structure from the dataplane. In Elastic Trie (ET), we seek for a solution where with limited memory accesses, the dataplane program can find the IP responsible for the traffic cluster and send the related information with a digest packet to a controller.

Optimization for network management. Dividing the time in fixed intervals simplifies the network events detection. At the end of each time window, 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 monitoring typically operate by exporting counters or specific data structures, e.g., sketches, to the controller at fixed time scales [38, 39, 53]. 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, 6]. If this last condition is met, solutions like dynamic routing of heavy flows [6, 20, 45] or dynamic flow scheduling [47] can be implemented. However, state-of-the-art solutions adopt a fairly large reporting time (typically 20 seconds [39, 48]) to overcome the limitations shown in §3, thus limiting network reaction capabilities. Instead, we propose to start tracking a coarse-grained approximation of the prefix responsible for our supported network events and iteratively refine it over the time. Then, depending on user settings, the controller receives a finer or coarser prefix information with bounded time delay.

Historical network trend awareness. Change detection is the process of identifying flows that contribute the most to traffic change over two consecutive time intervals [12]. Previous solutions [28, 39] rely on the controller to compute the differences from multiple intervals. With ET instead, we seek a solution capable to directly compute such an operation within the dataplane at the expense of minimal memory consumption. This allows to minimize as much as possible the amount of data to be exported.

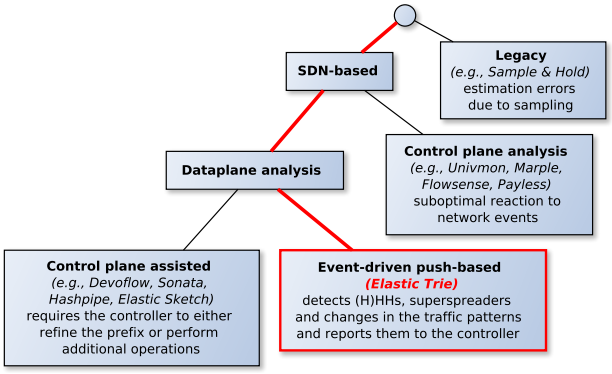


Figure 4: Design space in traffic aggregates detection.

5 ELASTIC TRIE

ET enables the detection of network events associated with high-volume traffic clusters from within the dataplane without the need to be coordinated by a controller. It works in a packet-driven manner and can be implemented using match-action based architectures such as RMT.

In this section, we describe how ET works taking HHH detection as an example and show that it can also be used to spot superspreaders and network traffic changes. We then discuss the user interface exposed from an ET-enabled switch to a network operator and show how ET can be used in the context of network-wide monitoring.

5.1 Data Structure & Algorithm

Let us use hierarchical heavy hitter (HHH) detection as an example to explain how ET works. In this scenario, we need to take into account packets (or bytes) to identify an aggregate. When we designed ET, we decided to use a tree-based data structure for the following reasons: (1) IP addresses are naturally organized according to prefixes into a hierarchy and (2) if aggregates are indexed in a tree, then by using standard longest-prefix matching techniques, it is possible to quickly find the small memory block where the prefix is stored, and then create the digest packet.

Each node of the prefix tree (trie) consists of three elements: the counter associated to the left child, the one associated to the right child and a timestamp. The counters represent the amount of traffic, i.e., packets, bytes or flows, 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 prefix $*$. The idea behind the proposed solution is to have a trie that grows or collapses to focus on the prefixes that account for traffic aggregates.

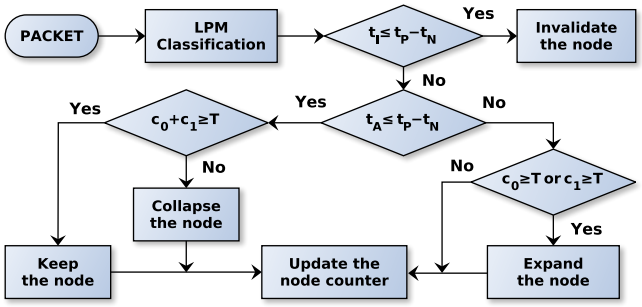


Figure 5: Flowchart showcasing input packet processing of the Elastic Trie detection algorithm.

To achieve this, we defined two timers: *active timeout* t_A and *inactive timeout* t_I, where $t_A < t_I$. The active timeout t_A is the upper bound interval after which the prefix is evaluated and possibly reported as (H)HH to the controller. The inactive timeout t_I defines instead the interval after which the node is considered inactive and its counters outdated. This configuration has the advantage that ET is not limited to a fixed time window and nodes are expanded and invalidated asynchronously: when using sketches, the whole data structure has to be zeroed by the controller at the end of each time window, this is not the case for ET. Fig. 5 depicts the key steps of ET algorithm. For every incoming packet, the LPM (corresponding node) is looked up. Let us denote by c_0 and c_1 the left and the right child counters of the node. The node timestamp (t_N) is compared against the packet arrival timestamp (t_P). Based on the comparison, node counter values and timeouts t_A, t_I, there are five possible cases to be considered:

Invalidating the node. If the inactive timeout t_I has expired ($t_I \leq t_P - t_N$), then the node has been inactive for a long time. The values of the counters are outdated and not relevant anymore for the detection. This happens when the source stops sending packets for a while. Because the detection process is built on a packet-driven basis, such situation cannot be evaluated in a different way. The inactive timeout mechanism handles the situation when the packets belonging to a source prefix start to flow again and the old values must be invalidated. Fig 6a 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 ($t_P - t_N < t_A < t_I$), but one of the counters (let us assume, for example, c_0) exceeds the threshold T that is used to discriminate heavy prefixes ($c_0 \geq T$). In this case, the subprefix associated with c_0 is (optionally) reported to the controller as HH but not as HHH yet. Fig. 6b depicts this case: the data structure automatically starts the refinement of the prefix (`10*`) by creating a new child (`100`) corresponding to c_0. According to the definition of hierarchical cluster, the original c_0 must be set to zero to remove the contribution of the newly created descendant. Since, we do not have any records for the newly created child yet, its node will have the timestamp set to the current packet timestamp and both its counters set to zero.

Keeping the node. This is the case when the inactive timeout t_I has not expired, but the active timeout t_A has ($t_A \leq t_P - t_N < t_I$), and the sum of counters exceeds the threshold T ($c_0 + c_1 \geq T$), but none of the counters contributes enough to reach the threshold individually ($c_0 < T$; $c_1 < T$). The case is shown in Fig. 6c. The prefix $11*$ is a HHH, because it exceeds the threshold T and none of its children exceed to threshold individually. The node is reported to the controller, its timestamp updated with the packet timestamp value and the counters are reset.

Collapsing the node. If the inactive timeout t_I has not expired, the active timeout t_A has expired ($t_A \leq t_P - t_N < t_I$) and the sum of counters does not exceed the threshold T ($c_0 + c_1 < T$), the node is collapsed (Fig. 6d). The node (`10*`) is removed from the structure, and it is replaced by the nearest parent. The counters of the parent node ($1**$) are zeroed and the timestamp set to the current packet timestamp. This is in contrast with the node invalidation case, where the nearest parent is not reinserted or renewed. Note that collapsing a node can happen only when the node has either none or one child because a node with two children does not match LPM.

Updating the node counter. This is performed when both the active and inactive timeouts have not expired ($t_P - t_N < t_A < t_I$) and none of the counters exceed the threshold T ($c_0 < T$; $c_1 < T$). In this scenario, the counter corresponding to the packet subprefix is updated 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 counters are updated instead of the current node counter.

5.2 Elastic Trie with the other events

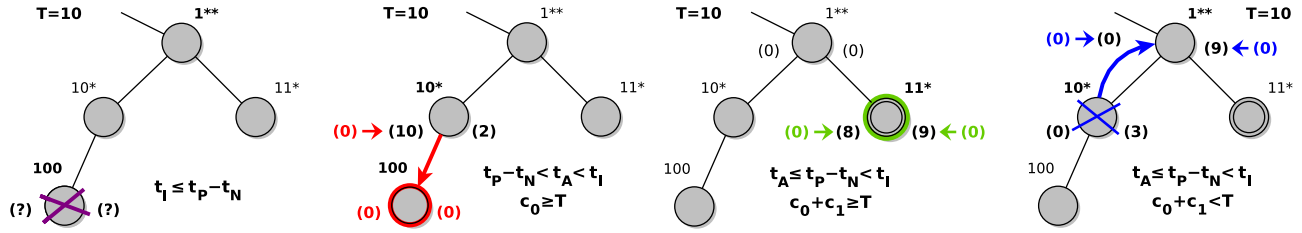
In this section we show how ET supports also the detection of superspreaders (DDoS victims) and network traffic changes.

Superspreaders detection. As introduced in §2, SS is a host that contacts at least a given number of distinct targets. Thus, to enable such a detection, 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], a memory-efficient probabilistic data structure commonly used to test for set membership. Specifically, we deployed the filter 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 that adjust the hierarchical structure is the same, but a test-and-set operation on the filter is performed for each incoming packet and the node counters are updated only if a new unique flow is detected.

Change detection. Changes in the traffic patterns can be detected by modeling the normal traffic behavior based on the past history and looking for significant changes that are inconsistent with the model itself. By tracking the number of nodes expanded or collapsed over an active timeout interval t_A, it is possible to spot sudden changes.

We added an integer counter which is incremented 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. Thus the counter value should vary around zero. When it is not the case, significant changes in the short-term traffic behavior have happened and are promptly reported to a controller.



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

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

Limitations. A single ET instance cannot track HHH and SS at the same time. To detect both events, it is necessary to deploy two individual structures side-by-side. While the HHH detection needs to use counters to track the number of packets, the SS detection tracks unique flows. On the other hand, the change detection can be freely combined with both packet or flow aggregates and built independently on top of HHH or SS detection trie.

5.3 User Interface

ET can detect the network events discussed in this paper without the need of any specific query from the operator. When an event is detected, a digest message is sent from the switch to the controller. It is still possible though to alter the behavior of the dataplane by changing T and t: recall that a traffic cluster is defined as the IP prefixes that contribute with a traffic volume, in terms of either bytes, packets or flows, larger than a threshold T during a time interval t.

Level of aggregation. The first parameter is the threshold (T). It directly impacts the number of reported prefixes and the memory requirements of the data structure, as shown in the evaluation section (§7): the higher the threshold, the less prefixes will be reported – they aggregate more traffic.

Tree building speed. The second parameter is the time interval (t). This allows the operator to effectively control the speed at which the tree is built and events reported. As shown in the evaluation section (§7), low values negatively affect the memory requirements of the data structure but allow the operator to react in a more timely fashion. On the other hand, high values allow ignoring transient events. As described in detail in the algorithm section (§5.1), we, introduce two time intervals parameters: *active timeout* t_A and *inactive timeout* t_I to distinguish between node reporting and invalidation time. In §5.4 we further define variable active timeout mechanism which beneficially allows setting different intervals for different tree levels.

5.4 Discussion

In this section we discuss an optimization that accelerates the trie building mechanism, thus speeding up the detection process, and show how ET could be used to perform network-wide monitoring operations.

Variable active timeout. The starting condition for the structure is associated to a trie composed by the zero-length prefix $*$. Depending on the packet flow, the trie is then built to focus on the heavy prefixes. Although the refinement process, as explained in §5.1, does not depend on the selected active timeout, the process of deciding if a specific prefix is a traffic aggregate and the potential reporting to the controller does. This means that in the worst case scenario a full IP address is reported after $32 \times t_A$ 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 intervals and corresponding thresholds for different prefix lengths, i.e., smaller timeout and threshold for shorter prefixes and vice versa.

Network-wide monitoring. The digests received by different ET-enabled switches can be used by a central controller to perform network-wide traffic analysis. Besides the obvious network-wide heavy hitter detection [29] use case, which is inherently possible by setting the appropriate threshold T in the switches and aggregating the received notifications in the control plane, others are also feasible. Specifically, the superspreader notifications received by different switches can be used by a central controller to perform the degree histogram estimation [46], commonly adopted practice to detect botnets involved in coordinated scans [26]. Furthermore, the controller can leverage the (H)HH primitive with an appropriate threshold T to detect global network icebergs [57]. Those are particularly difficult to detect within a single system, as the responsible packets might come from a large number of hosts and thus traverse different paths. We leave the evaluation of our system in a network-wide context to future work.

6 IMPLEMENTATION

This section discusses the implementation of ET on programmable hardware, using FPGA and the $P4_{16}$ language [18].

6.1 P4 prototype

Fig. 7 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 classification stage, (B) the main memory and bloom filter used to gather traffic statistics alongside related timestamps and (C) the control logic to dynamically adjust the hierarchical data structure and to report results of the detection to an external controller.

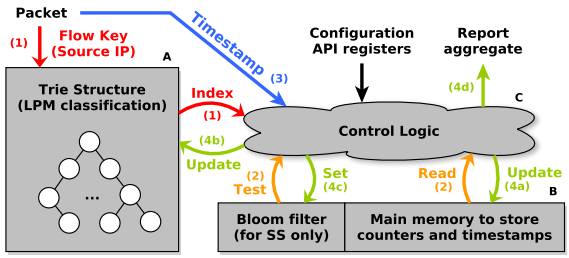


Figure 7: Elastic Trie dataplane architecture.

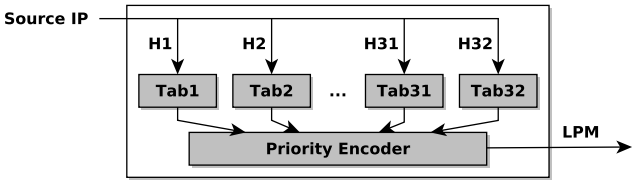


Figure 8: The LPM classification stage in P4.

Each incoming packet is first parsed to extract the desired flow key, i.e., source IP[3]. Then, the hierarchical tree structure is accessed to find the LPM (step 1). The result of this stage is an index that is used to access the main memory, where the data structure of the associated node is stored (step 2). Optionally for SS also an item presence is tested in the Bloom filter. The read values are compared with the packet timestamp (step 3) and the user settings, i.e., T, t. The appropriate operation is computed (step 4) following the specifications described in the previous section. Specifically, the comparison can trigger an update of the main memory (step 4a), an update of the LPM classification scheme (step 4b), setting an item in the Bloom filter (step 4c) or a push notification to the controller (step 4d). In the following, we provide a more detailed description of the mapping between the three main building blocks and $P4_{16}$ match-action constructs.

LPM classification stage. Although P4 offers built-in match tables supporting LPM, we could not utilize them for implementation of the trie structure, since the latest P4 specification does not support modifications 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 used a hash table for each prefix length (Fig. 8), thus requiring 32 hash tables to support each IPv4 prefix[4]. Each hash table is implemented as a register array. Upon packet arrival, all the hash tables are read in parallel, by hashing 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 preallocated to each hash table, we use a direct access based only on the prefix value itself (the `IDENTITY` algorithm in P4 API) for some of the shortest prefix tables. Each hash table lookup result can then be represented as a single bit value, `1` (found), `0` (not found) respectively. We then concatenate these bits to form a bitvector, which serves as input key for a static ternary match table implementing 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 counters and a timestamp. We use 32-bit wide packet counters and 48-bit wide timestamp as it is available in the packet metadata structure in P4. To detect hash collisions 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). In the case of a hash collision, the nearest shorter prefix node is used. The Bloom filter can be also easily implemented in P4 as a combination of a register-based bit array and a set of hash functions.

Control logic. This stage compares the packet timestamp with the node timestamp and applies the logic described in §5.1. The node collapse or expansion is performed by updating the appropriate hash table storing the specific prefix that needs to be adjusted, while the push-based mechanic is implemented by generating a packet digest (`digest` extern object in P4 API) containing the IP prefix detected as traffic aggregate alongside its node information such as the sum of the counters and the timestamp.

6.2 Potential limitations and challenges

Although we successfully implemented ET in P4 and compiled the code in the behavioral model [17], this does not guarantee ET being efficiently implementable in any high-speed P4 target available today. As briefly mentioned in §3, the main constraint of programmable switches is the number and the pattern of accesses to on-chip registers. Unfortunately, at the time of working on the paper, we did not have access to one to provide a full implementation on an actual hardware target. This section thus discusses the adopted pattern of accesses to on-chip registers, the potential limitations of the ET data structure and challenges in porting our code to commercially available P4-enabled switches.

The core part of the ET algorithm uses a register array to store the nodes of the tree. In the worst-case scenario, ET requires at most only two memory reads and writes (reading and updating a single node and optionally reading and updating its parent or child node). This could be a problem if the target does not support multiple accesses into the register array. However, the array can be eventually split into two parts, to store nodes in even and odd levels of the tree separately. This trick can guarantee a single read/write pattern per packet.

Another specific drawback of the ET algorithm is the need for a custom LPM implementation, which requires 32 hash tables to support each prefix length. Hash tables are implemented as a single register array requiring 32 reads and at most one write (inserting or removing a node in the tree), in contrast to the tree nodes, each

[3]While Elastic Trie is oblivious to the specific packet field used as flow key, the source IP address is commonly used for SS and (H)HH detection.

[4]Using 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.

Chip	LUTs		Regs	Frequency
	Logic	Memory		
Virtex 7	11 088	2 880	14 104	172.4 MHz
Virtex US+	9 135	2 641	14 103	307.9 MHz

Table 2: HW resources used and frequency achieved by Elastic Trie FPGA implementation.

hash tables stores only a single bit information. Using the same trick the array can be optionally split to have a register for each table. If a P4 target enables only a limited number of independent reads in a single stage, hash table registers can be eventually separated into more consecutive stages.

Finally, the custom LPM implementation forces spreading of the ET algorithm across multiple pipeline stages and thus introduces undesirable read/write dependencies, e.g., the memory index acquired from the LPM hash tables is used to access the node information which can result back in updating the LPM hash tables in case of a node collapse or expansion. In fact, the LPM hash table registers need to be accessed from the first stage to acquire the index, but also from the second stage to write the updates. However, this could potentially be a challenge for some of today's P4 targets if a set of registers is strictly bound to a single stage, effectively forcing the compiler to put all the registers in a single stage.

6.3 FPGA prototype

To demonstrate the general feasibility of implementing ET data structure on existing programmable targets and to quantify its requirements in terms of hardware resources, we at least created a pure VHDL implementation of ET for two different FPGAs, i.e., Xilinx Virtex 7 (model XC7VH580T) and Virtex UltraScale+ (model XCVU7P). We merged the LPM stage and main memory records into one memory block and utilized distributed memory to implement it as 32 parallel hash tables. Tab. 2 shows the chip occupancy and frequency of our design for both platforms. The latency introduced by the design is 7 clock cycles. Considering the achieved frequency, the latency is 40.6 ns for Virtex 7 and 22.75 ns for UltraScale+. The architecture is capable to process a new packet every 4 clock cycles (the first 3 out of 7 stages are pipelined). This results in an overall throughput of 43.10 Mpps for Virtex 7, and almost twice as much, 76.97 Mpps, for the UltraScale+ platform.

7 EVALUATION

Following a common practice adopted in evaluating P4 enabled solutions [39, 48, 52], we implemented a C++ simulation model to assess our approach against real traffic traces from an ISP backbone network. Additionally, the two FPGA prototypes (§6.3) provide insights about expected performance on real hardware and by compiling the P4 code in the behavioral model [17], we verified its correctness. In this section, we first describe our setup and we evaluate the trade-offs of ET. Then, we discuss its detection accuracy against the supported network events (Tab. 3 summarizes them all). We also evaluate ET varying memory occupancy and data structure configuration and compare it with state-of-the-art solutions. Finally, we analyze the controller-dataplane communication overheads and our detection speed.

Traces. We used four different one-hour packet traces from CAIDA [9, 10] recorded from 10 Gbps backbone links in San Jose and Chicago in 2009 and 2016 (both directions A and B). Each of the trace contains between 1.6 and 2.4 billion packets with mean transmission rates in range 440k-640k pkts/s (2.3-3.9 Gb/s) and flow rates up to 61k flows/s. The traces are distributed in one minute chunks and each chunk contains on average 30M packets with around 840K unique IP addresses. For further and detailed statistics for the individual CAIDA traces we refer to [11]. Unless otherwise stated, all the following results in the section are indicated as an average evaluated over the continuously replayed chunks of all four CAIDA traces. Unfortunately, we could not use the newer datacenter traces from the Facebook Network Analytics Data Sharing program [1], as they are sampled. Other publicly available datacenter traces from 2009 [5] have been anonymized without prefixes being preserved, which also makes them inappropriate for the type of tests needed in this paper.

Setup. We first set the primary measurement reporting time (active timeout t_A in our case) to 20 seconds and the inactive timeout t_I to 5 minutes. Then, when assessing the variable active timeout behavior (discussed in §5.4), we set it differently for each trie level. Specifically, we used an exponential function that specifies the value of the timeout for each of the trie level: the closer the node to the root, the lower the timeout. This allows to have much smaller timeouts for shorter prefixes, thus enabling a quicker tree build. In fact, the refinement process does not directly depend on the selected active timeout. It can be better understood as an upper bound for the delay between two reports, especially when there are no changes in detected aggregates. During the refinement process new aggregates are always reported immediately when the threshold is exceeded, thus the real reporting time is effectively much smaller and proportional to the rate of the threshold being reached. 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).

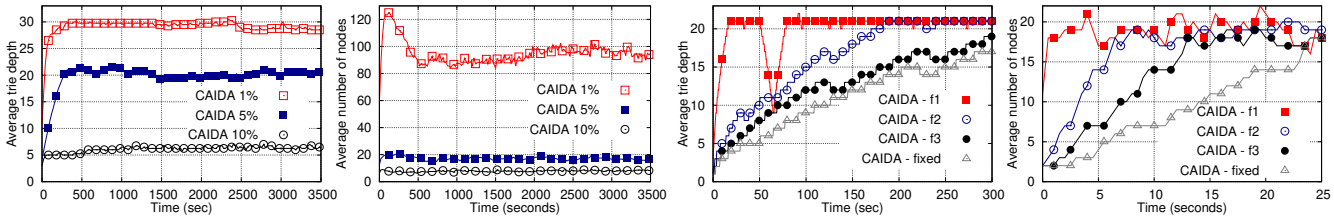
Metrics. We evaluated the number of required nodes and the trie depth varying the configuration parameters. Then, to estimate its network event detection capabilities, we used the F_1 *score* metric [52]. Assuming T_P (*true positives*), F_P (*false positives*) and F_N (*false negatives*), $F_1 = 2T_P/(2T_P + F_P + F_N)$. Unless otherwise stated, the F_1 score is always indicated as the average over the chunks of the traces.

Prefix comparison. We define two ways assessing ET detection capabilities: (1) can ET report the exact prefix? (2) can ET report at least a 2 bit shorter (coarser grained) version of the prefix? As by construction, ET starts reporting an approximation of the responsible prefix and iteratively refine it over time, we believe this is a good metric to better grasp the trade-off between fast detection and accuracy.

Memory allocation. ET has been architected to be implemented in hardware where allocation or de-allocation of memory at runtime is not possible. Thus, we statically pre-allocate a specific amount of memory for our data structure. An invalidation or collapsing of nodes is mostly used to track the traffic changes, not to manage the

Network event	Event definition	Implementation using Elastic Trie	Management tasks
(Hierarchical) Heavy Hitters	Identify hosts/prefixes which contribute with a traffic volume more than a defined threshold T during a time interval.	Node counters to count volume of specific prefixes. Expand and keep node events to identify exceeding the threshold T.	accounting, traffic engineering
Superspreaders	A superspreader is a host that contacts at least a given number of distinct destinations over a short time period (spread detection applied to source hosts).	Bloom filter to identify distinct destinations. Node counters to count distinct destinations. Expand and keep node events to identify source prefixes exceeding a predefined threshold.	scans, spread of malware detection
DDoS victims	A DDoS victim is a host that is contacted at least by a given number of distinct sources (spread detection applied to destination hosts).	Bloom filter to identify distinct sources. Node counters to count distinct sources. Expand and keep node events to identify destination prefixes exceeding a predefined threshold.	DDoS detection
Changes in traffic patterns	Identify hosts/prefixes which contribute the most to the traffic changes over the last time interval.	Tracking a difference of number of expanded and collapsed nodes to detect this event. Expand, collapse node events to identify specific prefixes/hosts involved.	anomaly detection, DoS detection

Table 3: Implementation of different network events detections using Elastic Trie algorithm.



(a) Fixed active timeout, varying threshold T. **(b) Variable active timeout varying function.**

Figure 9: Trie depth and number of nodes varying threshold and timeout behavior.

memory. The lack of memory, thus, results only in worse detection accuracy due to more hash collisions.

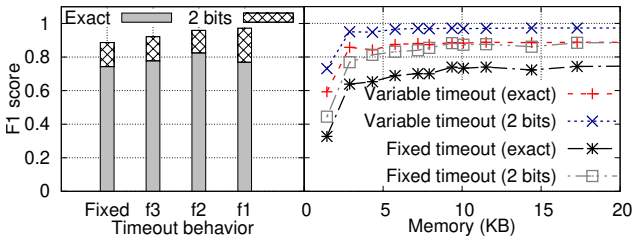
7.1 Data structure properties

Fig. 9a shows ET's average depth and average number of nodes over time for CAIDA traces varying the 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 learning phase of the trie at the beginning of the trace, when the trie has to build up from the less specific prefix. After this phase, the trie reaches a steady state that reflects the current traffic behavior. Fig. 9b offers, for the threshold of 5%, a more detailed view on the learning phase and compare the impact of variable active timeout with different exponential 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. In contrast, aggressive functions are much more sensitive to traffic patterns, resulting in potential fluctuations of the trie.

7.2 HHH detection

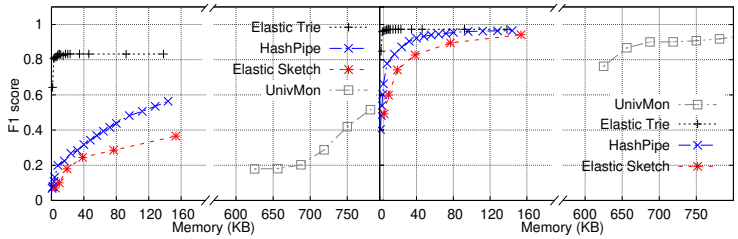
In this section we first present the HHH detection capabilities without resource constraints, then our implementation driven results. The former does not take into account the impact of implementation details such as amount of available memory or potential hash collisions. This allows us to get an understanding of the behavior of our solution in the best case scenario. The latter takes into account limitations in memory availability, as well as hash collisions that might happen during the classification stage. This allows us to get an understanding of the trade-offs between memory and detection results.

Results without resource constraints. Fig. 10a shows the HHH detection capabilities. We used a threshold of 5% and generated the results using both exact and relaxed prefix comparison. Since the basic behavior of ET is to build a trie that focuses on the prefixes that account for a large share of the traffic, sometimes it might happen that due to a transient event the system does not have enough time to finalize the building process and to fully identify the responsible traffic aggregate. However, reporting a partial and approximate result can still be very useful for the network operator. Indeed, the figure shows that the accuracy with exact prefix detection is significantly lower than its only 2 bit coarser grained prefix version. The effect of variable active timeout can also be seen in Fig. 10a. When using a more aggressive variable timeout the F_1 score increases, due to a smaller false negative rate. In contrast, the precision decreases causing a higher false positives rate. This is a direct effect of smaller active timeouts that lead the system to detect more prefixes including the previously mentioned short-term



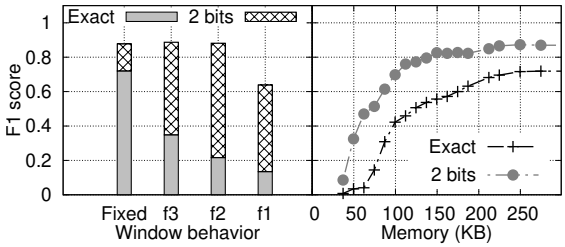
(a) Variable active timeout. **(b) Available memory.**

Figure 10: HHH detection capabilities.



(a) Exact prefix comparison. (b) Relaxed prefix comparison.

Figure 11: Comparison between Elastic Trie, HashPipe, Elastic Sketch and UnivMon of (H)HH detection capabilities.



(a) Variable active timeout. (b) Available memory.

Figure 12: Superspreader detection capabilities.

aggregates. In this scenario, the F_1 score is always between 0.9 and 1.0. Using different functions for variable active timeout, it is then possible to fine tune the trade-off between recall and precision to maximize the F_1 score metric.

Implementation driven results. We assess the impact of the amount of available memory over the F_1 score. We find that our solution can successfully detect, with approximately 0.75 F_1 score, the exact HHH prefix using a fixed active timeout and less than 20KB (Fig. 10b). If a coarser grained prefix is accepted, which is less precise by only two bits, then the F_1 score jumps to 0.9. 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. When using a variable timeout (Fig. 10b), the results improve sensibly. In this case, it is possible to detect the exact HHH prefix with 0.85 F_1 score with less than 8KB. Moreover, if a 2 bit coarser grained HHH prefix is accepted, the F_1 score jumps to 0.98. Increasing the available memory does not significantly improve the detection capabilities of the system, because it is bounded by the ability of the trie to react and build up according to the input traffic patterns.

In Fig. 11, we compare the HHH detection capabilities of ET against state-of-the-art solutions: UnivMon [39], Elastic Sketch [52] and HashPipe [48]. All of them are able to detect HH only as full length prefixes (addresses). Moreover, 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 amount of traffic they are actually sending. To perform 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 above the fixed threshold T.

Fig. 11a shows the results using exact prefix comparison. To reach a F_1 score around 0.5-0.6, HashPipe needs a lower amount of memory (~144KB) than Elastic Sketch (~320KB), which is still much lower than the amount needed by UnivMon (~800KB). In contrast, ET significantly outperforms other solutions. This is also confirmed by the results obtained when a coarser grained prefix is permitted (Fig. 11b). Nevertheless, the memory requirements of the four solutions represent a fair comparison metric. HashPipe, Elastic Sketch and ET have similar memory requirements, but HashPipe can only detect Heavy Hitters, while our solution enables, at the same time, also traffic pattern changes detection. UnivMon is not restricted to a single network event, but requires 90% more memory to work. Finally, Elastic Sketch in its heavy flows mechanism ignores mice flows. In contrast, ET can, thanks to the adopted trie-based data structure, aggregate these flows into shorter prefixes, which results in more accurate HHH detection.

7.3 Superspreader detection

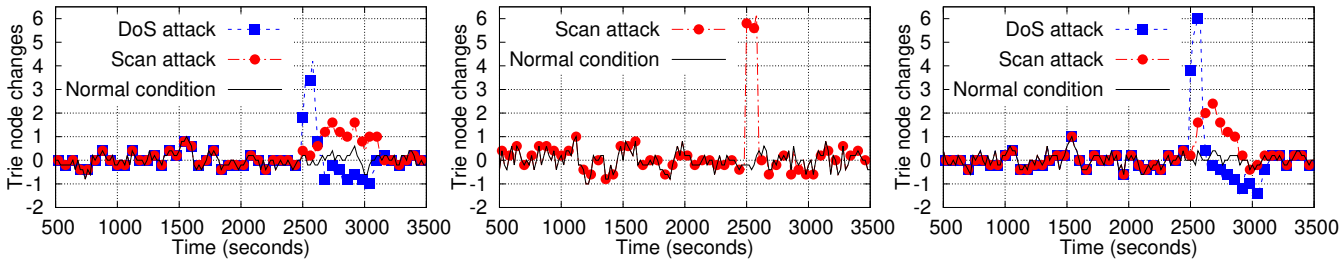
As in the HHH case, we first introduce the results without taking into account available memory or hash collisions. Then, we show the trade-offs between available memory and superspreader detection capabilities.

Results without resource constraints. Fig. 12a shows the superspreader detection capabilities without the impact of available memory or hash collisions using CAIDA traces and varying the active timeout behavior. In contrast to the same evaluation carried for the HHH detection, the results show that the system performs better using a fixed timeout. In this case, it is clear that the trie cannot build in time, as F_1 score grows sensibly when we use a 2 bit coarser grained superspreader prefix. Overall, for fixed active timeout the detection capabilities are still good, as F_1 is around 0.9.

Implementation driven results. In Fig. 12b 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, 25KB of preallocated memory for prefix trie structure, and we varied the amount of memory available for the Bloom filter. We find that superspreaders can be successfully detected with an F_1 score of approximately between 0.72 and 0.87 with less than 250KB of allocated memory. Among the considered solutions already available in the literature, i.e., UnivMon, Elastic Sketch and HashPipe, only the former theoretically supports superspreader detection. However, its current implementation does not allow to reproduce such a test. We were thus unable to compare it against our system.

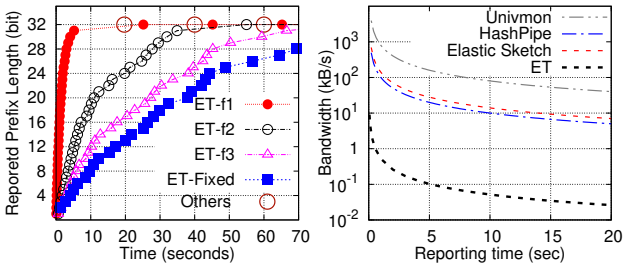
7.4 Change detection

To demonstrate the traffic change detection capabilities of ET, we artificially injected network traffic simulating a sudden heavy flow and a scanning into one of the CAIDA traces. The attack has been emulated after 2500 seconds since the beginning of the trace. A sudden HH and scanning are two types of attacks that can potentially change traffic patterns. At the same time, they are also pretty different: while a HH is typically a source that sends a huge amount of traffic to a designated destination, the scan is a source contacting many random destinations. Fig. 13a shows the time on the x-axis and a moving average of trie changes (difference between number



(a) Fixed active timeout, (H)HH. (b) Fixed active timeout, superspreaders. (c) Variable active timeout, (H)HH.

Figure 13: Change detection capabilities varying active timeout and trie building behavior.



(a) Detection speed. (b) Bandwidth utilization.

Figure 14: Comparison of detection speed and bandwidth utilization between ET and other approaches.

of expanded and collapsed nodes) on the y-axis. Note that tree is built based on HHH detection using a fixed active timeout of $t_A = 20$ seconds. In the figure we can distinctly see differences during normal conditions and the state under DoS attack or scan. In Fig. 13b, we repeated the same test but from a different perspective. Now the trie is built on top of the SS detection. In this case, the HH is not detected at all, because it represents a communication with only one distinct destination. On the other hand, the scan, as a typical case of superspreader, is much more significant now. Finally in Fig. 13c we run again the same test and build the trie for HHH detection using the variable active timeout mechanism. Due to the accelerated trie construction, there are many more changes in the trie over a short time period. This allows to highlight small changes in the traffic patterns, as shown when comparing the scan behavior for Fig. 13a and Fig. 13c.

7.5 Controller-dataplane communication

In this section, we asses how fast our data structure can provide useful results to an external controller alongside the involved overheads. We used the same setup as in the previous change detection case, and injected a sudden HH into a CAIDA trace. However, this time we focused on the events reported to the controller, especially on the delay to detect the first coarse-grained approximation of the prefix, and later the final prefix responsible for the attack. Fig. 14a shows the prefix length reported on y-axis with respect to the time from the beginning of the HH on x-axis. It can be seen that immediately after the start of the HH the coarse grained prefix is reported and then continuously refined over time. The figure also shows the effect of a variable active timeout. When using the most aggressive variable timeout function, the final responsible prefix is detected in less than 5 seconds, and then reported regularly every 20 seconds. The active timeout parameter can thus be understood as an upper bound for the delay between two reports. If there is no change detected when the timeout expires, the current prefix will be reported again. In the figure, we also include the reports generated periodically by state-of-the-art solutions such as Univmon, Elastic Sketch and HashPipe. Although, an external controller can be instructed to retrieve those data structures at shorter timescales, it is important to dimension the statistic retrieval process coherently with the actual time needed to get the hardware information (as shown in Fig. 3). Specifically, in the case of Univmon, Elastic Sketch and HashPipe, consecutive requests cannot be lower than the time needed to retrieve the data from the hardware. This effectively creates a lower bound in the detection speed for those systems. The same does not apply for ET, as it does not need the external controller to retrieve the full data structure to detect a certain event.

Finally, Fig. 14b compares the amount of data exchanged between a single switch and an external controller when either ET, Univmon, Elastic Sketch and HashPipe are in place. Varying the reporting time window (x-axis), we calculated the required bandwidth assuming 800KB size for the Univmon data structure, 140KB for the Elastic Sketch data structure and 100KB for the HashPipe data structure (default values according to respective papers). For ET we set 12B as the size of a single prefix report, which is enough to report the prefix, the length and also the associated sum of counters and timestamp. We then calculated the average number of HHH reports per second from CAIDA traces running ET with a threshold of 5 %. The figure shows that in comparison to the other solutions, ET can save a significant amount of control plane traffic (more than two orders of magnitude).

8 RELATED WORK

In the past many SDN-based monitoring solutions which rely on OF-based statistics retrieval from switches have been proposed [15, 49, 53]. They suffer from important limitations: (1) coupling between forwarding and monitoring rules, (2) the controller needs to know in advance which flows have to be monitored in the dataplane and (3) as the dataplane exposes just simple counters, the controller needs to do all the processing to determine the network state. In contrast, our solution reports to the controller the network events of interest as soon as they happen, without the need of central coordination. To speed up the identification process, iterative

refinement of the traffic of interest is done directly in the dataplane, to avoid expensive control plane interactions, contrary to solutions in [32, 41, 56]. Although algorithms that use iterative refinement of flows to determine heavy hitters [55] and anomalies [34] have been proposed in the past, they were not targeted for match-action type architectures. Dynamic trie-based data structures have been used for many years [19, 23, 33]. However, in contrast to them, our solution is time-based and hardware-friendly. The main difference is the way that nodes are expanded and invalidated based on stored timestamps, so the event reports are not limited to periodic time windows.

More recently, a number of monitoring frameworks leveraging P4 programmability have been proposed [30, 38, 39, 44, 48, 52]. FlowRadar [38] keeps track of all the flows in the network, their associated counters, and exports this information periodically to a remote collector, which ultimately uses them for various monitoring applications targeted to datacenters. In contrast, our aim is to offload as much as possible the controller, by directly exporting processed traffic information. UnivMon [39], ElasticSketch [52] and SketchLearn [30] use sketch-based data structures in the dataplane to record network traffic statistics and export them at fixed time intervals to the control plane which is in charge to perform a number of measurement tasks. Although some of these solutions apply optimizations to compress as much as possible the data structure, as demonstrated in §3, the interaction between control and dataplane can be very expensive and imply delays that are not acceptable. HashPipe [48] determines the top-k heavy hitters in the dataplane, while Popescu et a. [44] presents a solution for hierarchical heavy hitters detection. They both cover one single measurement task, while our solution is more generic. Finally, Sonata [28] 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 refinement is done by the controller, while in our case, directly in the dataplane.

9 CONCLUSION

We proposed a push-based approach to network monitoring, where the dataplane informs the control plane only when specific conditions are met. To achieve this, we presented a new data structure, Elastic Trie, that enables the detection of traffic pattern changes and either (hierarchical) heavy hitters or superspreaders 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 architectures 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 either (hierarchical) heavy hitters or superspreaders, and at the same time by looking at its growing rate it is possible to identify changes in the traffic patterns. We prototype our solution in P4 and for two different FPGA targets. From our FPGA implementations we provide information about expected performance on real hardware and from our C++ model we show that Elastic Trie achieves high accuracy in detecting the targeted events with the memory constraints imposed by today's switches.

ACKNOWLEDGMENTS

We thank our shepherd Mina Tahmasbi Arashloo and Ran Ben Basat, Theophilus Benson and anonymous reviewers for their valuable comments that helped us to improve the paper. This work was supported by the National Programme of Sustainability (NPU II), project IT4Innovations excellence in science – LQ1602, by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 co-funded by the Ministry of Education, Youth and Sports of the Czech Republic and European Regional Development Fund and by the UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL project (EP/P025374/1).

REFERENCES

[1] 2018. Data Sharing on traffic pattern inside Facebook's datacenter network. https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/.
[2] 2018. sFlow. http://www.sflow.org/about/index.php.
[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX.
[4] Ran B. Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *International Conference on Network Protocols (ICNP)*. IEEE.
[5] Theophilus Benson. 2010. Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
[6] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2011. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *COnference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.
[7] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. In *Communications of the ACM (CACM), Volume: 13, Issue: 7*. ACM.
[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
[9] CAIDA. 2018. Anonymized Internet Traces 2009 – 17th September 2009. http://www.caida.org/data/passive/passive_2009_dataset.xml.
[10] CAIDA. 2018. Anonymized Internet Traces 2016 – 17th March 2016. http://www.caida.org/data/passive/passive_2016_dataset.xml.
[11] CAIDA. 2020. Statistical information for Anonymized Internet Traces. http://www.caida.org/data/passive/passive_trace_statistics.xml.
[12] Christian Callegari, Stefano Giordano, Michele Pagano, and Teresa Pepe. 2012. Detecting Anomalies in Backbone Network Traffic: A Performance Comparison Among Several Change Detection Methods. *International Journal of Sensor Networks, Volume: 11, Issue: 4.*
[13] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the Microburst Culprits with Snappy. In *Workshop on Self-Driving Networks (SelfDN)*. ACM.
[14] Baek-Young Choi, Jaesung Park, and Zhi-li Zhang. 2003. Adaptive random sampling for traffic load measurement. In *International Conference on Communications (ICC)*. IEEE.
[15] Shihabur R. Chowdhury, Md. Faizul Bari, Reaz Ahmed, and Raouf Boutaba. 2014. PayLess: A low cost network monitoring framework for Software Defined Networks. In *Network Operations and Management Symposium (NOMS)*. IFIP/IEEE.
[16] B. Claise. 2018. Cisco Systems NetFlow Services Export Version 9. https://tools.ietf.org/html/rfc3954.
[17] P4 Language Consortium. 2018. P4 Switch Behavioral Model. https://github.com/p4lang/behavioral-model.
[18] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification, version 1.0.0. http://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.
[19] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. 2008. Finding Hierarchical Heavy Hitters in Streaming Data. In *Transactions on Knowledge Discovery from Data, Volume: 1, Issue: 4*. ACM.
[20] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling Flow Management for High-performance Networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
[21] Nick Duffield, Carsten Lund, and Mikkel Thorup. 2001. Charging from Sampled Network Usage. In *Internet Measurement Workshop (IMW)*. ACM.
[22] Cristian Estan, Ken Keys, David Moore, and George Varghese. 2004. Building a Better NetFlow. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[23] Cristian Estan, Stefan Savage, and George Varghese. 2003. Automatically inferring patterns of resource consumption in network traffic. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[24] Cristian Estan and George Varghese. 2002. New Directions in Traffic Measurement and Accounting. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[25] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. 2001. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Transactions on Networking, Volume: 9, Issue: 3*. IEEE/ACM.

[26] Yan Gao, Yao Zhao, Shobha V. Schweller, Yan Chen, Sawn Song, and Ming-Yang Kao. 2007. Detecting stealthy attacks using online histograms. In *International Workshop on Quality of Service (IWQoS)*. IEEE.

[27] The P4.org Applications Working Group. 2018. In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report.pdf.

[28] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[29] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Symposium on SDN Research (SOSR)*. ACM.

[30] Qun Huang, Patrick P. C. Lee, and Yungang Bao. 2018. Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[31] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[32] Lavanya Jose, Minlan Yu, and Jennifer Rexford. 2011. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*. USENIX.

[33] Nils Kammenhuber and Lukas Kencl. 2005. Efficient statistics gathering from tree-search methods in packet processing systems. In *International Conference on Communications (ICC)*. IEEE.

[34] Faisal Khan, Nicholas Hosein, Chen-Nee Chuah, and Soheil Ghiasi. 2011. Streaming Solutions for Fine-Grained Network Traffic Measurements and Analysis. In *Architectures for Networking and Communications Systems (ANCS)*. IEEE Computer Society.

[35] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. 2003. Sketch-based Change Detection: Methods, Evaluation, and Applications. In *Internet Measurement Conference (IMC)*. ACM.

[36] Anukool Lakhina, Mark Crovella, and Christophe Diot. 2004. Diagnosing Network-wide Traffic Anomalies. In *Computer Communication Review, Volume: 34, Issue: 4*. ACM.

[37] Anukool Lakhina, Mark Crovella, and Christophe Diot. 2005. Mining Anomalies Using Traffic Feature Distributions. In *Computer Communication Review, Volume: 35, Issue: 4*. ACM.

[38] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[39] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[40] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. 2006. Is Sampled Data Sufficient for Anomaly Detection?. In *Internet Measurement Conference (IMC)*. ACM.

[41] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2014. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[42] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[43] Noviflow. 2018. Noviswitch 1132 product guide. https://noviflow.com/wp-content/uploads/NoviSwitch-1132-Datasheet-V2.0.pdf.

[44] Diana Andreea Popescu, Gianni Antichi, and Andrew W. Moore. 2017. Enabling Fast Hierarchical Heavy Hitter Detection Using Programmable Data Planes. In *Symposium on SDN Research (SOSR)*. ACM.

[45] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[46] Vyas Sekar, Michael K. Reiter, and Hui Zhang. 2010. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *Internet Measurement Conference (IMC)*. ACM.

[47] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[48] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Symposium on SDN Research (SOSR)*. ACM.

[49] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. 2010. OpenTM: Traffic Matrix Estimator for OpenFlow Networks. In *Passive and Active Measurement (PAM)*. Springer-Verlag.

[50] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. 2005. New Streaming Algorithms for Fast Detection of Superspreaders. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society.

[51] Yinglian Xie, Vyas Sekar, David A. Maltz, Michael K. Reiter, and Hui Zhang. 2005. Worm Origin Identification Using Random Moonwalks. In *Security and Privacy (SP)*. IEEE Computer Society.

[52] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[53] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. 2013. FlowSense: Monitoring Network Utilization with Zero Measurement Cost. In *Passive and Active Measurement (PAM)*. Springer-Verlag.

[54] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[55] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. 2007. ProgME: Towards Programmable Network Measurement. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[56] Ying Zhang. 2013. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM.

[57] Qi (George) Zhao, Mitsunori Ogihara, Haixun Wang, and Jun (Jim) Xu. 2006. Finding Global Icebergs over Distributed Data Sets. In *Principles of Database Systems (PODS)*. ACM.