Efficient Detection of Flow Anomalies with Limited Monitoring Resources

Jalil Moraney, Danny Raz
Computer Science Department
Technion - Israel Institute of Technology
{jalilm, danny}@cs.technion.ac.il

Abstract—**Real time detection of flow anomalies is a critical part of wide range of management and security applications in many Cloud and NFV systems. Solutions based on per-flow records have become impossible due to the increasing traffic volumes and the limited available resources such as TCAM entries and fast counters.**

In this paper we study a novel dynamic control mechanism that allows detecting flow anomalies using only a limited number of counters. Starting from the simple observation that it is impossible to guarantee instantaneous detection of flow anomalies with a limited amount of counters, we study the trade-off between the time required to detect the anomaly and the number of available counters.

We implemented the scheme in an OpenFlow enabled switch, where the logic is implemented in the controller, and demonstrate that it can be used to detect a single flow anomaly within large real traffic volume. To further reduce the detection time, we also implemented the scheme logic inside the switch and used the controller only for configuration. This implementation indeed yielded a faster detection and lower monitoring communication overhead while not introducing any significant observable costs at the switch itself.

I. Introduction & Motivation

Network traffic monitoring is a critical building block in various management, control and security applications. Traditionally, network monitoring tools collect per-flow traffic information that can be stored locally or polled to a centralized management station and analyzed. This analysis provides important information like trends in network load and utilization, performance of traffic engineering systems, or security vulnerabilities. In current networks, more and more network analytics tools process this information online to generate real time alerts regarding performance or security issues.

Solutions based on per-flow records, which is the case for most monitoring tools existing today, are not adequate anymore due to the increasing traffic volumes in many cloud and NFV scenarios and the limited available resources such as TCAM entries and fast counters. Contrary to popular belief, the shift into SDN will not resolve this issue. Indeed, software is more flexible and more counters can easily be provisioned in a softswitch, however, the performance burden on the VM running the software device will make it hard or even impossible to support large number of counters in real time. Thus, there is a vital need to detect flow anomalies with a limited number of counters.

The term "flow anomaly" is very broad and it covers any deviation from the normal traffic characteristics; this includes malicious traffic such as a Distributed Denial-of-Service (DDoS) attack, heavy hitter - flow that uses an unusual large portion of the available bandwidth, or an unusual increase in the demand for a certain network service. In this paper we study the ability to detect these flow anomalies in an SDN setting where the number of counters is substantially smaller than the number of flows.

It is not hard to observe that when the number of counters is smaller than the number of (active) flows, one cannot guarantee the instantaneous exact identification of all flow anomalies. One way to approach this situation is by giving up accuracy and identify a set of flows that contains the anomaly as done for example in [1]. Our approach is different; we want to exactly identify the anomaly flows, but are willing to give up on the instantaneous requirement. That is, we allow a trade-off between the number of counters and the time it takes to identify an anomaly but we insist on exact identification of the problematic flows. In fact, as we show in this paper, the actual time to detect an anomaly in realistic scenarios is around 3 seconds while the needed number of counters is a small constant independent of the number of active flows.

We start with a formal definition of the problem and study the trade-off between the number of counters and the number of rounds[1] needed to identify a problematic flow in this theoretical framework. We then develop a provable accurate monitoring algorithm that can exactly identify a problematic IPv4 flow in a constant number of rounds.

We implement this algorithm on top of *Open vSwitch* [2], where the algorithm's logic is executed in the SDN controller. That is, at each round the controller polls the counters from the switch, reports a problematic flow if identified, decides on the new rules to be deployed, and configures the switch accordingly.

This solution requires a considerable communication overhead, and does not scale when the controller controls many network elements. Thus, we also implemented an additional solution where the monitoring algorithm logic is embedded inside the vSwitch code. For that, we extended the OpenFlow protocol [3] to allow the controller to configure the switch with the appropriate parameters as needed.

We then evaluated the performance of the algorithm using

[1]A round (or an epoch) is an iteration where counters data are collected and analyzed.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 610802 (Project CloudWave).

978-3-901882-85-2 © 2016 IFIP

these distinguished two methods. The evaluation is done by injecting network anomalies of various types[2] into CAIDA real life traffic trace [4], and replaying them into the SDN network. Our evaluation shows that one can detect an anomaly as fast as 3-4 seconds after it first appears using only a few counters (regardless of the number of flows) and with very little overhead. The evaluation also shows that implementing the monitoring logic inside the switch indeed yielded a faster detection and lower monitoring communication overhead, while not introducing any significant observable costs at the switch itself.

To best of our knowledge this work is the first one focusing on the trade-off between the amount of available resources and the time required to exactly identify an anomaly in the SDN domain. We go all the way from a theoretical analysis to the details of the implementation, tying together the fundamental abstract aspects with system issues related to the specific implementation and deployment. Thus, providing interesting and relevant insight regarding this important problem.

In Section II we define the problem and in Section III we introduce our algorithm and prove its characteristics. In sections IV and V we consider possible deployment schemes and discuss implementation details. The setup of the experiments and the performance evaluation are described in Section VI. In Section VII we survey related work and in Section VIII we present short conclusions.

II. The Monitoring Problem

We focus on detecting flow anomalies that appear in real situations in a specific network node using only a limited number of counters. For example, *Heavy Hitter* flows, are flows (i.e., an aggregated set of packets identified by certain fields in the header) that exceed a specified volume. DDoS attack, close enough to the victim, is actually a heavy hitter flow aggregated by the IP destination address, and the threshold is proportional to the maximal bandwidth the victim can handle.

A slightly different kind of network anomalies are the one defined by ratio rather than value. In these anomalies the ratio of certain type of "offending" packets out of all packets in the flow is higher than a characteristic threshold. One example of ratio based anomaly is SYN flood attack, in which the attacker tries to cause a resource exhaustion at the victim system. This is achieved by opening as many as possible TCP sessions without ever completing the three-way handshake, in order to keep resources bound to the half open sessions and preventing the system from opening legitimate sessions. For normal real TCP traffic the ratio of SYN packets in a flow (also known as SYN Arrival Rate - SAR) is about 0.07 (see [5]).[3]

To model the problem, we partition the time into constant length discrete segments (called epochs or rounds) and assume that the flow rates are fixed within each epoch but can change arbitrarily between consecutive epochs. We also assume that at most n unique flows are going through the network node of interest. The amount of usage (e.g., the actual number of packets) of the flows in this node at (the end of) epoch t is represented by a series of n real-valued variables, $\{u_i(t)\}_{i=1}^{i=n}$. We are given a series of n fixed real-valued per-flow limits, $\{l_i\}_{i=1}^{i=n}$, and say that *over usage condition* holds at epoch t if there exists i such that $u_i(t) > l_i$.

Under an appropriate setup, the problem of detecting flow anomalies can be reduced to the problem of detecting over usage condition. The detection of heavy hitter anomaly can be modeled as detecting over usage condition of a flow, when the flow is aggregated by a specific header field and the flow's limit is the maximal traffic allowed for that header field. The detection of ratio based anomalies, requires calculating the ratio of offending packets out of all packets in a flow and comparing this ratio against the characteristic ratio. Thus, it can be modeled as detecting over usage condition of a flow, when the flow's limit is the characteristic ratio. That being said, this approach is not able to detect more complex forms of flow anomalies that can not be reduced to detecting over usage conditions.

As mentioned, we are interested in detecting *over usage condition* by any of the flows, using only a bounded number of counters, m, which is usually far smaller than the number of (active) flows, n. To achieve that, in each epoch we assign to each of the counters a subset of the variables u_i to measure. We denote by $S_j(t)$, the subset of variables measured by counter j at epoch t and by $Y_j(t)$ the aggregated usage value of the subset S_j at the end of epoch t, i.e., $Y_j(t) = \sum_{u \in S_j(t)} u(t)$.

We define a *monitoring algorithm* to be an algorithm which in each epoch t:

1) Obtains the measurement data of the previous epoch, i.e. the values $Y_1(t-1), Y_2(t-1), ..., Y_m(t-1)$.
2) Decides on the new assignment of flows to counters (the subsets $S_j(t+1)_{j=1}^{j=m}$). This is done based on the obtained measurement values, the previous assignment (the subsets $S_j(t)_{j=1}^{j=m}$) and the monitoring strategy.
3) Checks whether an over usage condition occurred according to the obtained measurements and reports it.

A monitoring algorithm is *k-correct* if it is able to detect any over usage condition that holds for at least k consecutive epochs. We say that a monitoring algorithm is *immediately correct*, if it is 1-correct. The motivation of this definition is that an immediately correct monitoring algorithm is able to detect over usage of any flow as it occurs, however this may require a large number of counters. Furthermore, we say that a monitoring algorithm is *eventually correct*, if it is k-correct for some $k \geq 1$. This allows the relevant management application to relax the instantaneous detection requirement and dramatically reduce the number of required counters.

We do not allow false negative as we want to detect all anomalies. However, in some cases it makes sense to have false positive alerts. A monitoring algorithm is *p-accurate*, if its false positive rate (FPR) is less than p. We say that it is *accurate*, if it is 0-accurate. In this terminology, the

[2]We present results for SYN flood attacks and for heavy hitters.

[3]The theoretic characteristic ratio depends on the exact anomaly in concern, for SYN flood attack the SAR is always smaller than quarter, since in a legitimate TCP connection there is one SYN packet and two non SYN packets completing the three-way handshake followed by at least one additional data packet.

algorithm is accurate in the sense of never detecting an over usage condition if it did not occur. We say that a monitoring algorithm is *ideal*, if it is both immediately correct and accurate.

It is clear that using $m \geq n$ counters, one can detect any over usage condition as soon as it occurs, by the straightforward ideal monitoring algorithm that simply assigns each counter to measure a distinct flow variable (thus n counters are sufficient), and at the end of each epoch checks that none of the counters passed the limit of the variable assigned to it.

When the number of counters is strictly smaller than the number of flows this algorithm cannot be used and the situation becomes more complex. For a small number of flows one can show that exactly n counters are needed. This is best illustrated when we are limited to use 1 counter to monitor 2 unique flows ($n = 2$), for any assignment of the counter there is an input which shows that the algorithm is not ideal.

More generally, it is possible to show that at least $m = \lceil \frac{n}{log(n+1)} \rceil$ counters are needed for ideal monitoring algorithm. Let consider a simpler problem where each flow usage is either 0 or 1, and we want to detect which flows have a value of 1. There are 2^n possible distinct states for the flows, each counter have $n+1$ values (0,1,...,n) and there are m counters, thus there are $(n+1)^m$ possible distinct states for the counters. In order to have ideal algorithm it easy to see that $(n+1)^m \geq 2^n$ must hold, since $(n+1)^m = 2^{log((n+1)^m)} = 2^{mlog(n+1)}$ we can take logarithm from both sides and end up with $m \geq \frac{n}{log(n+1)}$.

Thus, when the number of counters is smaller than the number of flows, we might not be able to guarantee immediate detection. A straightforward generalization of the trivial (assign each flow a counter) algorithm yields a $\lceil \frac{n}{m} \rceil$-correct, accurate monitoring algorithm. In this generalization, we split the flows into $\lceil \frac{n}{m} \rceil$ groups each of size m at most, and at each epoch we assign the counters to measure one of the groups. In order to guarantee the detection of any over usage condition that holds for $\lceil \frac{n}{m} \rceil$ epochs, we reassign the counters in a cyclic manner throughout all the groups. Note that in practice this requires the network device to reconfigure the counters very often which in many cases is associated with a considerable overhead.

III. The Monitoring Algorithm

In this section we introduce an eventually correct and accurate monitoring algorithm called "IPv4 prefixing", for the over usage detection problem. The algorithm itself is based on a variation of the MRT algorithm [6] presented in [7], where the main concept is to use *prefix-trie* to decide which flowsets to monitor in the next epoch.

In this approach, we identify each flow by a unique *string* above some alphabet and each flowset by a regular expression above the same alphabet, such that all flows contained in the flowset are the flows represented by the strings matching the flowset's regular expression. The motivation behind this approach, is to identify each flow by an IP address and each flowset by a CIDR mask, such that a flowset is the group of all flows that their corresponding IP address is included in the flowset's CIDR mask.

The basic idea behind the algorithm is as follows. At each epoch the algorithm monitors a set of disjoint flowsets, F; that is, for each flowset $f \in F$, it assigns a counter to measure the aggregated value of all flows contained in f. At the end of each epoch, the algorithm examines the values of the counters and classify flowsets to be either "interesting", "uninteresting" or "keep". Since the main target is to detect an over usage condition, we say that a flowset is interesting, if the current aggregated value of this flowset is higher than the minimal limit of each of its flows. Furthermore, we say that a flowset is uninteresting if the current aggregated value of this flowset is lower than half the minimal limit of each of its flows, and otherwise the flowset is keep.

When a flowset f is classified as interesting, the algorithm generates a partition of f, $refine(f)$. After generating the refined flowsets, the set F is modified to include the new flowsets and to exclude f. If a flowset was classified as interesting and it contains only one flow, then the algorithm reports this flow as "over using". This is correct since the classification of interesting flowsets is based on the minimal limit of all flows in it.

When a flowset is classified as uninteresting, we are interested in removing it from the monitoring at next epoch. Due to the requirement of disjoint flowsets in F, we can not simply remove the flowset from F, we need to remove its "siblings" too and add their "most recent" ancestor. Such folding should take place if and only if none of the siblings is classified as interesting in the current epoch. If there is a flowset classified as interesting, we should change the classification of its uninteresting siblings to keep.

If a flowset is identified as keep, it will stay in F unless at least one of its siblings is classified as uninteresting in the same epoch. The pseudo code of the algorithm is presented in Fig. 1.

The algorithm makes use of several auxiliary procedures to manage the assignment of the variables to the counters. These procedures are (1) $getCounter(f)$ which returns the index of the counter that is measuring the flowset f, (2) $assignCounter(f)$ which assigns the flowset f to be measured by an available counter, and (3) $freeCounter(f)$ which releases the counter measuring the flowset f, this is achieved by assigning ϕ to that counter. Additionally, the algorithm uses the notation of $flows(f)$ to describe the set of all flows contained in the flowset f, sometimes we also abuse the notation to describe flows of a set of flowsets.

The input to the algorithm is a set of disjoint flowsets called *root* flowsets, the algorithm is restricted to look for over usage condition only in flows contained in one these flowsets. The algorithm initialization step assigns counters only to these root flowsets. The specific behavior of the algorithm is determined by the definition of the following operations:

- $fineThreshold(f)$: which returns the threshold that if the aggregated value of flows in f is above it, then we should refine the flowset f.
- $foldThreshold(f)$: which returns the threshold that if the aggregated value of flows in f is below it, then we

```
Algorithm "IPv4 Prefixing"
Initialization:
$F \leftarrow roots$ and $\forall 1 \le i \le m, S_i(0) \leftarrow \phi$
for each $f \in F$ do $assignCounter(f)$ end for
At the end of each epoch $t$:
 1: Obtain the measurements from the counters for the current epoch, $y_1 \leftarrow Y_1(t), y_2 \leftarrow Y_2(t), ..., y_m \leftarrow Y_m(t)$.
 2: $interesting \leftarrow \{f \in F | c = getCounter(f), y_c \ge fineThreshold(f)\}$
 3: $uninteresting \leftarrow \{f \in F | c = getCounter(f), y_c < foldThreshold(f) \wedge siblings(f) \cap interesting = \phi\}$
 4: $toAdd \leftarrow \phi$
 5: $toRemove \leftarrow \phi$
 6: for each $f \in interesting$ do
 7:     if $|flows(f)| = 1$ then
 8:         Report $flows(f)$.
 9:     else
10:         $toAdd \leftarrow toAdd \cup refine(f)$
11:         $toRemove \leftarrow toRemove \cup \{f\}$
12:     end if
13: end for
14: for each $f \in uninteresting$ do
15:     $uninteresting \leftarrow uninteresting \setminus siblings(f)$
16:     $toAdd \leftarrow toAdd \cup fold(f, siblings(f))$
17:     $toRemove \leftarrow toRemove \cup \{f\} \cup siblings(f)$
18: end for
19: for each $f \in toRemove$ do
20:     $freeCounter(f)$
21: end for
22: for each $f \in toAdd$ do
23:     $assignCounter(f)$
24: end for
25: $F \leftarrow F \setminus toRemove \cup toAdd$
```

Fig. 1. eventually correct and accurate monitoring algorithm for IPv4 prefixes using m counters

should fold the flowset f.

- $refine(f)$: returns a set of flowsets which is a partition of f. i.e., $\forall g_1, g_2 \in refine(f), flows(g_1) \cap flows(g_2) = \phi$ and $flows(\bigcup refine(f)) = flows(f)$.
- $fold(G)$: returns a flowset which is the lowest common ancestor of the flowsets in G, i.e. $flows(fold(G)) = flows(\bigcup G)$.
- $siblings(f)$: returns a set of flowsets which were added to F in the same refine operation as f.

The pseudo code of these operations is described in Fig. 2.

We are interested in the properties of the algorithm in the most general case where the algorithm starts by measuring the flowset which includes all possible IPv4 flows in the switch, generally $roots = \{$"0.0.0.0/0"$\}$. If at the end of an epoch there is a flowset that needs refinement then it is performed by adding two flowsets that have bigger CIDR mask and differ by the least significant bit not covered by the mask. If at the end of an epoch there is a flowset that needs to be folded, then it is performed by removing the flowset and its sibling in favor of a flowset that have smaller CIDR mask and the common IPv4 prefix of both siblings. In order to guarantee accuracy, the fining threshold of a flowset should be the minimal limit of all flows included by that flowset.

Theorem 1. *Algorithm "IPv4 prefixing" with the "IPv4 prefixing operations" is* 32*-correct monitoring algorithm.*

Proof. Assume there is a flow f_i with limit l_i that is overusing for 32 epochs starting from epoch t, i.e., $u_i(t), u_i(t+1), ..., u_i(t+31) \ge l_i$. Since roots contains the flowset that contains all flows, it contains f_i too, then by the refinement process the set F at epoch t contains a flowset, g_1 such that $f_i \in flows(g_1)$. We note that $get\_CIDR\_mask(g_1) \ge 0$.

At every epoch starting from t till $t+31$, any flowset that contains f_i will get refined, since the refinement threshold is the minimal limit of all flows in the flowset and the aggregated usage of that flowset is higher than l_i, since $u_i \ge l_i$ in these epochs. Thus, there is a series of $g_1, g_2, ..., g_{32}$ such that $\forall j, f_i \in flows(g_j)$ and $g_{j+1} \in refine(g_j)$. Since $get\_CIDR\_mask(g_1) \ge 0$ and at each refinement the CIDR mask is bigger by one, there is $k < 32$ such that $get\_CIDR\_mask(g_k) = 32$, which means that $|flows(g_k)| = 1$ and therefore $flows(g_k) = \{f_i\}$ and f_i will be reported in $k \le 32$ epochs. ∎

Theorem 2. *Algorithm "IPv4 prefixing" with the "IPv4 prefixing operations" is accurate monitoring algorithm.*

Proof. Assume by contradiction that the algorithm reports a false positive, that is, it reports a flow f_i that does not have over usage condition. A flow is reported only if a flowset of size 1 containing it passed its fining threshold, however the fining threshold of a flowset of size 1, is the limit of the flow contained in it. Thus, in the epoch f_i was reported $u_i \ge l_i$ holds, in contradiction to the assumption of false positive. ∎

```
Algorithm "IPv4 prefixing operations"
 1: function fineThreshold(f)
 2:     return $min_{f_i \in flows(f)}\{l_i | l_i$ is the limit of flow $f_i\}$
 3: end function
 4: function foldThreshold(f)
 5:     return $\frac{fineThreshold(f)}{2}$
 6: end function
 7: function refine(f)
 8:     $CIDR\_Mask \leftarrow get\_CIDR\_mask(f)$
 9:     $IPv4\_address \leftarrow get\_IPv4\_address(f)$
10:     $f_1 \leftarrow createFlowset(IPv4\_address, CIDR\_mask + 1)$
11:     $f_2 \leftarrow createFlowset(IPv4\_address \oplus 0^{CIDR\_mask}||1||0^{32-(CIDR\_mask+1)}, CIDR\_mask + 1)$
12:     return $\{f_1, f_2\}$
13: end function
14: function fold(G)
15:     assert(all flowsets in $G$ have a common IPv4 prefix)
16:     $CIDR\_mask \leftarrow min_{g \in G}\{get\_CIDR\_mask(g)\} - 1$
17:     $IPv4\_address \leftarrow get\_common\_prefix(G) \& 1^{CIDR\_mask}||0^{32-CIDR\_mask}$
18:     return $createFlowset(IPv4\_address, CIDR\_mask)$
19: end function
20: function siblings(f)
21:     $CIDR\_Mask \leftarrow get\_CIDR\_mask(f)$
22:     $IPv4\_address \leftarrow get\_IPv4\_address(f)$
23:     return $createFlowset(IPv4\_address \oplus 0^{CIDR\_mask-1}||1||0^{32-CIDR\_mask}, CIDR\_mask)$
24: end function
```

Fig. 2. Definitions of the operations fineThreshold, foldThreshold, refine, fold, siblings for the general case of IPv4 prefixing

One of the most interesting aspects of the "IPv4 prefixing" algorithm, is the number of counters it uses to detect the anomalies. In order to estimate the number of counters, we model the monitoring process as a search problem in a graph, and show that each node in the search frontier requires using a counter by the algorithm. We also discuss how different factors affects the number of counters used. We show that this number depends heavily on the outcome of the $fineThreshold$ operation and the number of flows that exceed their limits.

For the sake of simplicity we first introduce the modeling of "IPv4 prefixing" algorithm with the "IPv4 prefixing operations" with a single root, i.e. $roots = \{$"0.0.0.0/0"$\}$. We model the search graph as a directed tree, where each node represents a flowset and an edge connecting node representing flowset f to node representing flowset g, if and only if $g \in refine(f)$. This modeling yields a binary tree with one root that represents the flowset $\{$"0.0.0.0/0"$\}$, and leafs that represents all flowsets of size one, i.e. single flows.

In this modeling the process of detecting a "violating flow", a flow that exceeds its threshold, is equivalent to searching down a path from the tree's root to the leaf representing that flow. The iterative nature of algorithm, requires maintaining a set of flowsets currently monitored, where each $refine$ or $fold$ operation changes the set of the next epoch. Since the algorithm assigns a counter to monitor each flowset in the current set, it is easy to see that the number of currently used counters is the size of the search frontier in the tree.

The generalization of this modeling to the case where roots set size is larger than one is straightforward. The main concept of searching down a path to the leaf representing the violating flow remains the same, however the search graph in this case is a forest graph of r binary trees rather than a single binary tree. The observation that the number of currently used counters by the algorithm is the size of the search frontier is still valid in this generalization, where the frontier spreads through the whole forest.

The algorithm "IPv4 prefixing" refines the monitoring of a flowset, if the current usage value is bigger than the minimal limit of all flows contained in it, this is crucial to guarantee accuracy as shown in Theorem 2. Thus, the outcome of the $fineThreshold$ effects the number of counters used in the next epoch directly.

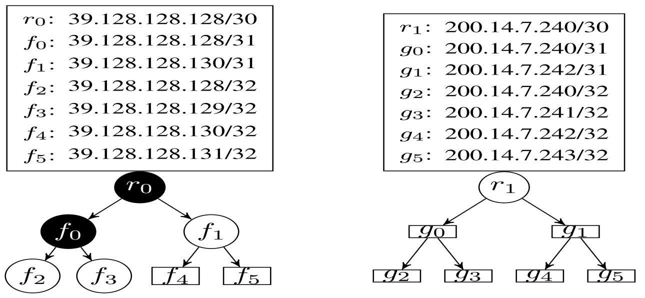
In order to analyze the effect of the given limits, and thus the $fineThreshold$ operation effect, on the number of counters used by the algorithm, one should consider the characteristic usage values of the flows.

If the given limits are bigger than the characteristic usage values, then the algorithm will search only down the path to the leaf of the violating flow. On the other hand, when the given limits are close to the characteristic usage values, the algorithm will search down all possible paths even if there are no violating flows. This is true, since the aggregated usage of any sub-tree will exceed its limit while no leaf will do.

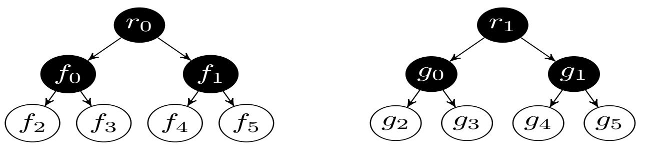
Fig. 3 describes the algorithm's frontier with roots set of size two. White circled nodes are nodes currently in the frontier (i.e., being monitored in current epoch), black circled nodes are nodes which were at previous epochs in the frontier and rectangled nodes are nodes not yet searched by the algorithm. Fig. 3a illustrates the frontier, when the limits are bigger than the characteristic usage values. While Fig. 3b illustrates the frontier, when the limits are close to the characteristic usage values.

In order to estimate the actual number of counters used, one should take into consideration also the existence of multiple violating flows. Since the number of paths the algorithm will search is effected by the number violating flows, and thus more counters are needed.

Let x be the maximal number of needed refinements operations to reach a single-flow flowset, e.g., $x = 32$ in the "IPv4



(a) limits higher than characteristic value


r_0 f_0 f_1 f_2 f_3 f_4 f_5
r_1 g_0 g_1 g_2 g_3 g_4 g_5

(b) limits close to characteristic value

Fig. 3. The algorithm's frontier with the roots $39.128.128.128/30, 200.14.7.240/30$

prefixing" algorithm with $roots = \{$"0.0.0.0/0"$\}$. Let c be the number of violating flows and $r = |roots|$.

When there is single violating flow ($c = 1$) and the given limits are higher than the characteristic usage values, the frontier of the search is of size exactly $(r - 1) + x + 1$, thus the number of needed counters is $(r - 1) + x + 1$. This value consists of $x + 1$ nodes in the forming path frontier and $r - 1$ nodes of the roots flowsets that does not contain the violating flow as seen in Fig. 3a with $r = 2$, $c = 1$ and $x = 2$.

If there are multiple violating flows ($c > 1$) and the given limits are higher than the characteristic usage values, without knowing the distribution of the violating flows we can only give an upper bound on the frontier's size. If we assume disjoint violating flows, then the upper bound is $(r - c) + (x + 1)c$.

The case where the given limits are close to the characteristic usage values, the algorithm will search down all paths and the forming frontier will contain all $r2^x$ leafs in the forest as seen in Fig. 3b. This loose estimate can be tightened with $min((x + 1)c, n)$; the minimal of either the number of nodes on paths from the roots to the violating flows or the total number of the flows monitored.

Note that in realistic scenarios the value of the counters keeps on changing throughout the process. This adds more noise into the systems and may create situation where more paths are searched.

It is worthy to note, that scaling the algorithm "IPv4 prefixing" to IPv6 networks is straightforward. It is clear the algorithm will become 64-correct while its accuracy guarantee and counters' estimates still hold.

IV. SDN Monitoring Schemes

In this section we study the actual implementation of the monitoring algorithm in an SDN setting. The theoretical model only assumes that the network nodes are capable of preforming the aggregated measurements but does not specify the management entity or whether the logic is executed locally at the network node or at a centralized management entity.

In SDN networks, a natural choice is to run the monitoring algorithm as an application on top of the SDN controller. In such a setup, the monitoring application should, at the end of each epoch: (1) *poll* the counters from the switches, (2) evaluate the received measurements and decide on counters' new assignment for the new epoch, (3) update the switches' counters with the new assignment and (4) report over usage conditions, if they exist, to a dedicated mitigation application also running on top of the SDN controller.

Note that this type of deployment imposes additional overhead in terms of network utilization by the control plan, i.e., the polling and update messages exchanged between the switches and the controller at every epoch. Another option is to embed the monitoring logic within the switch itself; this is possible since all decisions are based only on the switch's local data and the predefined monitoring algorithm. Thus, we consider two schemes: the *Polling Scheme* where the logic is implemented in the controller, and the *Pushing Scheme* where the monitoring algorithm is implemented as middleware on top of the switch itself.

In the Polling Scheme the monitoring application at the controller kicks in and installs in the SDN enabled network node counters' assignment that suits the $roots$ input set. For each flowset in the $roots$ set, the monitoring application sends a command to the node to assign a counter to measure all flows in this flowset.

At the end of each epoch, the monitoring application polls the value of each counter and based on the received value, the history and the algorithm state, decides on the classification of the flowset. If the flowset is classified as interesting, the monitoring application is responsible to send a command to the node to assign the counters for the new flowsets and free the counter used for the refined flowset. If a flowset is classified as uninteresting, the application sends commands to the node asking to free the counters of the flowset and all of its siblings and to assign a new counter for their ancestor. If a flowset is classified as keep, the application stores a local state for that flowset so that in the next epoch it will be able to subtract the saved value from the received value.

It is possible to implement the application without the need of saving a state per every currently monitored flowset. However, this requires to clear the counter of the flowset and then the received value at the end of the next epoch will be the exact usage value of the next epoch.

In the Pushing Scheme, the controller is still responsible for the control plane configurations and the network-wide installment of the forwarding rules. However, instead of deploying an active monitoring application at the controller, a passive application is deployed, responsible to configure the relevant nodes with the needed monitoring strategy in the switches. The strategy is determined by the exact monitoring algorithm and by the roots set to be monitored in each node. The actual implementation of these configuration commands is done using the "Experimenter" message in OpenFlow (see [8]).

The network's nodes are modified to host an application in the form of a *middleware*, responsible for receiving the monitoring configuration from the controller and installing the initial set of the counters at the node. The middleware then executes the monitoring algorithm locally on top of the network's node and configures the new set of counters at the end of each epoch accordingly.

When the middleware detects an over usage condition by one of the flows, it reports the flow to the controller. The controller is still responsible to handle the detected anomaly and to take network-wide actions, which depends on the application and might vary from changing the control plane to reconfiguring the monitoring goals of any of the network nodes.

V. Monitoring implementation inside an SDN node

We implemented both schemes on top of OpenFlow enabled software defined network, where we have one controller that uses *OpenFlow Protocol* [3] to communicate to a number of SDN enable switches.

OpenFlow provides an open protocol to set the flow-table in different switches by exploiting a common set of functions that vendors specific flow tables share. In OpenFlow enabled switch all packets are processed by the OpenFlow pipeline. Each flow table in the pipeline contains multiple flow entries, the protocol defines the structure of the entries and how incoming packets are matched against these entries.

Each flow entry consists of several components, the ones of interest are: (1) Match Fields - fields that indicate which data from the packet headers and its metadata should be matched against this entry. (2) Priority - a field indicating the matching precedence of the flow entry; packets will be matched against higher priority entries first till a match is found. (3) Counters - the following set of counters: *Received Packets, Received Bytes, Duration in seconds, Duration in nanoseconds*. (4) Instructions - a set of instructions that are executed when a packet matches the entry.

We exploit the fact that matching of packet is done first against higher priority flow entries. This allows the algorithm to install entries of refined monitored flowsets with strictly higher priority than entries of the original flowset, causing a matching at the refined entry only.

Our implementation keeps routing entries and monitoring entries separated. While routing entries could be installed on several tables, all of the monitoring entires resides at the same table. Furthermore, the routing entries forward matched packets to the monitoring table using the *Goto-table* instruction.

The refinement process of a flowset should yield a partition. Also, it should consider the number of entries needed to measure a single flowset, before simply incrementing the priority value. For example, in the case of detecting SYN flood attacks, the increment in the priority value should be two to allow installing two entries for each new flowset. More details are in [9].

VI. Evaluation

We tested our algorithm for detecting two kind of anomalies. The first is a bandwidth heavy hitter, a host sending high bandwidth traffic through the switch, and the second is a SYN flood attack. We use *Mininet* [10], [11] to emulate the software defined network and *Open vSwitch* [2], [12] as the OpenFlow enabled switch (a modified version was used to enable the pushing scheme). We also use the *Ryu* SDN framework [13] to build the monitoring application in Python.

The setup of the experiments is a single switch connected to the hosts via two different ports, and the routing entries were a simple forwarding traffic from one port to the other. *Tcpreplay* [14] was used to replay CAIDA traffic trace [4] into the network. Considering the prefixing nature of the algorithm, in order to keep the original characteristic of the real traffic and the privacy of the users, one should be careful to use real traffic traces that went through "prefix-preserving anonymization" such as the CAIDA traces.

At some randomly chosen point, after the start of replaying the traces, a host introduces a flow anomaly of the requested kind. In each of the experiments we measured, using wireshark [15], the number of epochs from the one in which the anomaly starts until its detection, the number of counters used and the number of OpenFlow packets exchanged between the controller and the switch. In the heavy hitter flow experiment, the host sent traffic at the rate of 20 Mbps while the per flow limits were set to 15 Mbps. In the SYN flood attack, a SYN packet was sent to the target through the switch for every four trace packets and the per flow limit was set to 0.03[4].

We would like to detect anomalies as soon as they occur; however the theoretical framework only deals with the number of epochs and not with the actual total time which depends on the setting of the epoch length. Fig. 4 compares the average numbers of epochs needed to detect a SYN flood attack in both polling and pushing schemes. The figure shows that when the epoch is longer than 0.1 seconds, both schemes are able to detect the attack in 32 epochs. This is expected since the "IPv4 prefixing" algorithm is $32 - correct$, so refinement of the offending flowset will occur in every epoch. The figure also shows that the shorter the epoch, more epochs are needed to detect the attack. Such behavior is expected since the shorter the epoch the harder it is to detect the current trend and the measurements values are inaccurate due to the normal traffic variability and accuracy of the time measurement. This behavior starts to appear in epochs of length 0.25 and 0.5 seconds, where the average number of epochs is still about 32, but with non-zero variance. In epoch of length 0.1 seconds, this behavior is more significant with average number of epochs of 47.5 and 53.4 in pulling and pushing schemes, respectively.

Clearly the polling scheme is much more sensitive to time precision, since the decisions are not taken locally and the actual measurement times are affected by the latency of the network. This is validated by the fact that the algorithm was not able to detect the attack at all in the polling scheme with epochs shorter than 0.1 seconds, while in the pushing scheme it did. The actual detection time is the multiplication of the epoch length by the number of epochs, and the shortest one was achieved in the pushing scheme with total time of 3.24 seconds. Although the detection took 64.8 epochs on average, due to the very short epoch of 0.05 seconds, the total time was still shorter than the time in the experiments with less epochs.

To better examine the detection time we compare in Fig. 5, the average time until detection for each anomaly type. The results show that for both types of the anomalies, pushing based setups were able to detect the anomaly with shorter epochs than polling based setups and that there is no significant difference when considering epochs longer than 0.5 second. There is a clear minimum value where the epoch length is the shorter one that still allows reducing the noise and making the measurements accurate enough. The maximal number of counters used was as expected; 33 for the heavy hitter case and 66 for the SYN attack case.

To stress the algorithm we contacted an experiment in which we concurrently launched four heavy hitters on top of the replayed network traffic. The source IP addresses of the anomaly flows, were chosen randomly from a subset of source addresses found in the replayed traces. Fig. 6 depicts the number of counters (monitoring rules in the switch tables) throughout the experiment. The number varies until addresses stop sharing a common prefix, and each host gets located fully.

The upper bound formula we provided on the number of counters was $(r - c) + (x + 1)c$, which yields an upper bound of 129 counters, with the appropriate parameters. Indeed, the maximal number of counters used at any epoch was 125 in the detection epoch, smaller than the predicted upper bound.

Next we examine the traffic overhead due to control messages. Fig. 7 shows the number of OpenFlow packets exchanged between the controller and the switch in both schemes. In the polling scheme, one can see that the shorter the epoch, the more OpenFlow packets are exchanged since more epochs are needed till the detection, and each epoch the controller sends commands to update flow entries in the switch. In the pushing scheme, the number of packets exchanged is stable for the short epochs. This is expected since there is no need to receive commands from the controller at every epoch. However, we note that the longer the epoch the more packets are exchanged, this is due to the OpenFlow protocol which requires an exchange of "echo packets" between a switch and its controller every 5 seconds. For instance, with epoch length of 20 seconds 95.77% of the exchanged packets were echo packets.

VII. Related Work

Several recent works have been focusing on efficient resource constrained flow monitoring, realizing that measurement is crucial for network management and control and even more so in the SDN domain [1], [6], [7], [16], [17].

Yuan et al. [6] presented the *flowset* concept to represent aggregated, which is an arbitrary set of flows defined according to the application requirements and/or traffic conditions. They also defined the *Flowset Composition Language* (FCL) to enable manipulation of flowsets in a formal manner. These definitions enabled the *Multi-Resolution Tiling* (MRT) algorithm they presented, which dynamically reprograms the

[4]Note that the measured SAR in the CAIDA traffic is about 0.014 and the SAR during a typical attack as reported in [5] is around 0.6.

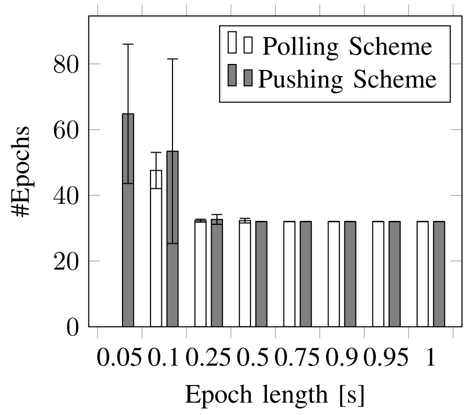


Fig. 4. Number of epochs needed to detect SYN flood attack

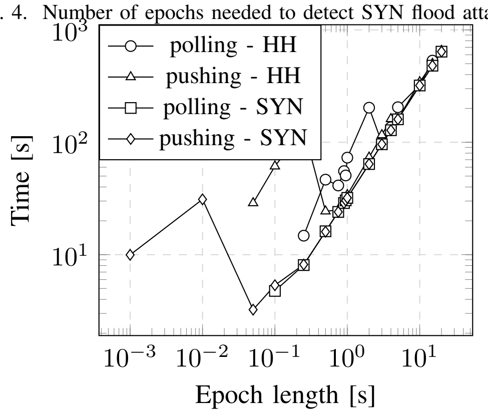


Fig. 5. Average time till detection

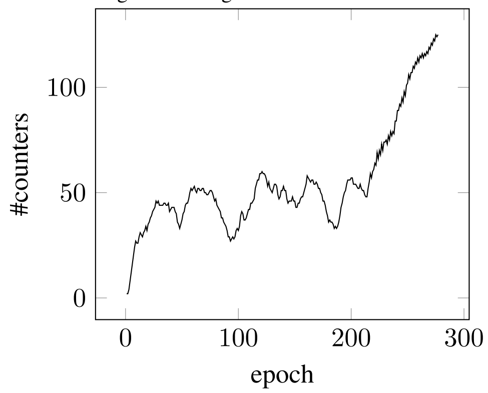


Fig. 6. The number of counters used by the algorithm at each epoch when detecting 4 concurrent heavy hitters

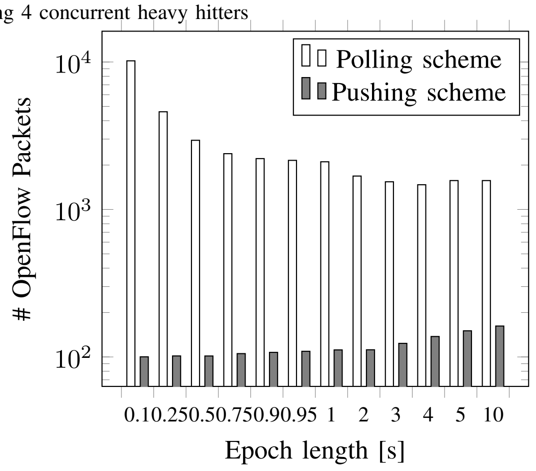


Fig. 7. Communication overhead

measurements to zoom in on "heavy hitters" by reallocating the associated counters.

One aspect of these works deals with the distributed nature of the network in which the same flow can be measured at different network nodes. The authors of [1] and more recently [7] proposed a network wise dynamic monitoring systems where a centralize management station (or the controller in the SDN case) dynamically configure monitoring rules in the different network elements. Such a centralized management entity can thus make use of global information in order to utilize the distributed monitoring resources (typically TCAM rules) in an efficient way, that is, getting as much precision as possible for the given monitoring resources.

In [16], Gangam et al. focus on combining information gathered from different nodes to generate a global picture; in their case they want to detect in real time what they call *global distributed icebergs*, a set of flows with a common property contributing little traffic at local nodes and a significant amount of traffic across the entire network. The main technical issue is how to define the local information to be collected in each node and how to decide which flows are globally interesting. Dilman and Raz [18] considered a similar setting but focused on communication overhead where the problem is to detect all global events while minimizing the amount of communication between the centralized station and the measurement nodes.

The field of Streaming algorithms is a theoretical framework dealing with online data analysis, which can be perceived as the flow measurement task in each node. Most of the papers in this field deal with theoretical or data based issues; a notifiable exception is the paper [19], where the authors consider maintaining statistics of a stream of data over time. They study the trade-off between the number of counters and the ability to collect and maintain this information.

Several recent papers mention the connection between the duration of the time data is stable and the ability to detect it [1], [17]. The authors of [1] also concentrated on a trade-off between the amount of available resource and the accuracy of the measurement, which is somewhat complementary to our study. Note, that like many of the works in this area, the authors of [1] do not present any analytical framework to study the trade-off and rather concentrate on system related issues.

VIII. Conclusions

The main contribution of this paper is the introduction of resource efficient flow anomaly detection algorithm that trade-offs the instantaneous detection requirement for the sake of full accuracy. Our results indicate that by allowing a small flexibility of 3-4 seconds in the instantaneous requirement, one can provide very accurate measurement with a very little monitoring resources. Furthermore, the results also imply that the execution of complex monitoring strategies on top of the switch itself are superior to the classic approach of SDN where the logic is executed at the controller. This is true when the logic requires only data from a single network element and when the relevant criteria are communication overhead and time to detect.

References

[1] M. Moshref, M. Yu, and R. Govindan, “Resource/accuracy tradeoffs in software-defined measurement,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13. New York, NY, USA: ACM, 2013, pp. 73–78. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491196

[2] “Open vswitch - production quality, multilayer open virtual switch,” http://www.openvswitch.org, accessed: 07-11-2015.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: Enabling innovation in campus networks,” *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[4] “CAIDA anonymized internet traces 2014 dataset,” http://www.caida.org/data/passive/passive_2014_dataset.xml, accessed: 07-11-2015.

[5] S. Nemade, M. K. Gurjar, Z. Jamaluddin, and N. N., “Early detection of syn flooding attack by adaptive thresholding (EDSAT): A novel method for detecting syn flooding based dos attack in mobile ad hoc network,” *International Journal of Advanced Research in Engineering & Technology (IJARET)*, vol. 5, pp. 79–86, 2014. [Online]. Available: http://www.academia.edu/7324329/EARLY_DETECTION_OF_SYN_FLOODING_ATTACK_BY_ADAPTIVE_THRESHOLDING_EDSAT_A_NOVEL_METHOD_FOR_DETECTING_SYN_FLOODING_BASED_DOS_ATTACK_IN_MOBILE_AD_HOC_NETWORK

[6] L. Yuan, C.-N. Chuah, and P. Mohapatra, “ProgME: Towards programmable network measurement,” in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM ’07. New York, NY, USA: ACM, 2007, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/1282380.1282392

[7] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, “Dream: Dynamic resource allocation for software-defined measurement,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14. New York, NY, USA: ACM, 2014, pp. 419–430. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626291

[8] “The openflow specefication v1.5,” http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf, accessed: 07-11-2015.

[9] J. Moraney, “Efficient Detection of Flow Anomalies with Limited Monitoring Resources,” Master’s thesis, Technion - Israel Institute of Technology, 2015.

[10] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[11] “Mininet an instant virtual network on your laptop,” http://www.mininet.org, accessed: 07-11-2015.

[12] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, “Extending networking into the virtualization layer,” in *Eight ACM Workshop on Hot Topics in Networks (HotNets-VIII), HOTNETS ’09, New York City, NY, USA, October 22-23, 2009*. ACM SIGCOMM, 2009. [Online]. Available: http://conferences.sigcomm.org/hotnets/2009/papers/hotnets2009-final143.pdf

[13] “Ryu SDN framework,” http://www.osrg.github.io/ryu, accessed: 07-11-2015.

[14] Tcpreplay - pcap editing and replaying utilities. [Online]. Available: tcpreplay.appneta.com

[15] Wireshark - network protocol analyzer. [Online]. Available: wireshark.org

[16] S. Gangam, P. Sharma, and S. Fahmy, “Pegasus: Precision hunting for icebergs and anomalies in network flows,” in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM) 2013, Turin, Italy, April 14-19, 2013*, 2013, pp. 1420–1428. [Online]. Available: http://dx.doi.org/10.1109/INFCOM.2013.6566936

[17] S. Meng, L. Liu, and T. Wang, “State monitoring in cloud datacenters,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1328–1344, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2011.70

[18] M. Dilman and D. Raz, “Efficient reactive monitoring,” *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 4, pp. 668–676, 2002. [Online]. Available: http://dx.doi.org/10.1109/JSAC.2002.1003034

[19] M. Datar, A. Gionis, P. Indyk, and R. Motwani, “Maintaining stream statistics over sliding windows: (extended abstract),” in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 635–644. [Online]. Available: http://dl.acm.org/citation.cfm?id=545381.545466