# NetFuse: Short-circuiting Traffic Surges in the Cloud

Ye Wang
Yale University

Yueping Zhang, Vishal Singh, Cristian Lumezanu, and Guofei Jiang
NEC Laboratories America

***Abstract*—Modern cloud and data center platforms suffer failures and performance degradation from large traffic surges caused by both external (*e.g.*, DDoS attacks) or internal (*e.g.*, workload changes, operator errors, routing misconfigurations) factors. If not mitigated, traffic overload could have significant financial and availability implications for cloud providers. In this paper, we propose NetFuse, a mechanism to protect against traffic overload in OpenFlow-based data center networks. NetFuse is (1) scalable because it uses passively-collected OpenFlow control messages to detect active network flows; (2) accurate because it uses multi-dimensional flow aggregation to determine the right criteria to combine network flows that lead to overloading behavior; and (3) effective in limiting the damage of surges while not affecting the normal traffic because it uses a toxin-antitoxin-like mechanism to adaptively shape the rate of the flow based on application feedback. Experimental results on a real OpenFlow testbed show that NetFuse is effective in identifying and isolating misbehaving traffic with a small false positive rate ($< 9\%$).**

## I. Introduction

Modern cloud and data center platforms comprise thousands of servers running complex applications.Although constructed with redundant network topologies [1] and well-engineered protocols [2], these platforms still suffer catastrophic failures and performance degradation when traffic overload occurs. Traffic overload is caused by external factors, such as DDoS attacks, but also by seemingly harmless internal factors, such as small workload changes, simple operator tasks, or routing misconfigurations. For example, Amazon EC2 was down in April 2011 due to a routing misconfiguration that mistakenly rerouted high-volume external traffic into the low-capacity internal network [3].

There is a large amount of research on identifying and mitigating network problems caused by an overabundance of traffic, falling into two categories: proactive and reactive. Proactive solutions propose to modify the data center architecture, protocols, resource placement, or applications to limit the occurrence of failures [4], [2], [5], [6], [7], [8]. However, they cannot easily cope with overloading behavior caused by unpredictable interactions between applications or with the infrastructure, as manifested in the Amazon incident. Reactive solutions focus on how to quickly detect faults after they occur [9], [10]. These approaches must constantly monitor network traffic and application behavior and may not scale when the price to gather measurements is high.

In this paper, we explore a mechanism that exploits the capabilities of OpenFlow switches [11] to prevent *data center network overloading problems*. OpenFlow is increasingly deployed in data centers to support diverse performance- or reliability-based application requirements [12]. In OpenFlow networks, switches connect to a logically centralized controller and notify it, using control messages, when new flows arrive or expire.

We propose NetFuse, a mechanism to protect against *cascading* network overloading with *little measurement overhead*. NetFuse sits between the switches and controller of an OpenFlow network and guards the network against the effects of traffic overload, similarly to how fuse boxes protect electrical circuits from surges. NetFuse leverages OpenFlow's capabilities to achieve lightweight network-wide overload detection and reaction, in three steps: (1) It uses OpenFlow control messages (*e.g.*, PacketIn, FlowMod, FlowRemoved) to *detect the set of active network flows*. Relying on control traffic, which notifies the controller of network events, such as flow departures and arrivals, eliminates the need for on-demand, expensive monitoring. (2) To detect overloading behavior, we employ a multi-dimensional flow aggregation algorithm that automatically determines an optimal set of clustering for the active flows (*e.g.*, based on VLAN, application, path, rate) and *identifies suspicious flows* for each criteria. (3) Finally, NetFuse limits the effect of these flows by adaptively changing their rate using a toxin-antitoxin mechanism.

We implement NetFuse and deploy it in a real OpenFlow network as a proxy between the switches and the controller. Preliminary experiments show that NetFuse detects network overloading caused by routing misconfigurations or DDos attacks with small false positive rates (less than 9%).

## II. Flow Aggregation

The first goal of NetFuse is to identify the flows with suspicious behavior. The definition of "suspicious" varies according to different contexts. For example, both excessively frequent DNS queries and extremely high traffic volume represent suspicious behavior. In practice, network operators need to specify overloading behaviors based on the operational experience and domain knowledge. Because no single flow may be suspicious, NetFuse performs flow aggregation to identify clusters of active flows with overloading behavior.

### A. Current Practice: Single-Dimensional Aggregation

**General problem** We formulate the general flow aggregation problem. The input is a set of $n$ flows: $\{f_1, f_2, ..., f_n\}$. Each valid flow aggregation can be modeled as a *set partition* $P$ (*e.g.*, $P = \{F_1 = \{f_1, f_3, f_5\}, F_2 = \{f_2, f_n\}, \ldots, F_k\}$). Our goal is to find the aggregations that exhibit overloading.

To capture the overloading behavior of an aggregation, we define its overloading score $S(P)$ as $(\max\{F\} - m^-\{F\})/m^-\{F\}$, where $max$ is the maximum rate across all aggregated flows in the aggregation and $m^-$ is the low median. Different aggregations have different scores, as in some

aggregations the overloading behavior shows more evidently than others. For example, in Figure 1, aggregation $P_1$ has a higher score than $P_2$ due to the sharpness of its peaks. The aggregation that best reveals overloading behavior is given by $P^* = \max_P S(P)$.

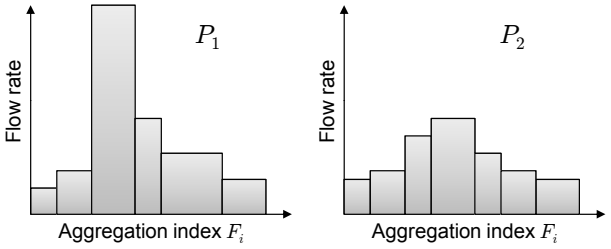



Fig. 1. Example of different flow aggregations.

Flow aggregation is a combinatorial optimization problem and NP-hard. Machine learning based clustering algorithms may be applied, but they require heavy training data and complex computation, making it unsuitable for online problem detection. In fact, it may not even be necessary to solve this general flow aggregation problem, because an arbitrary flow clustering may not be mapped to a practical overloading reason and thus a practical solution cannot be found. Even worse, it is also possible that a group of perfectly legitimate flows are labeled problematic only because they collectively show certain *statistical* overloading behavior.

| | Aggregation condition | Example overloading reason |
|---|---|---|
| P1 | ingress and egress | end-to-end flooding |
| P2 | source subnet | compromised VMs |
| P3 | destination subnet | flash crowd to specific VMs |
| P4 | destination port | attack against specific services |
| P5 | routing | routing misconfiguration |
| P6 | start time range | correlated data transfers |
| P7 | frequency threshold | new traffic load |
| P8 | duration threshold | short/failed connection attempts |
| P9 | burstiness threshold | buggy customized TCP |

TABLE I
EXAMPLE REASONABLE FLOW AGGREGATIONS.

**Reasonable aggregations** Instead of traversing arbitrary flow aggregations, we design NetFuse to enumerate a list of *reasonable* aggregations and find the one that best explains the observed overloading phenomenon. A reasonable aggregation corresponds to one or more *practical overloading reasons*. Table I exemplifies a list of reasonable flow aggregations and the corresponding overloading reasons. For example, $P_4$ aggregates the flows with the same destination port number together. If the overloading is caused by attacks against HTTP, the aggregated flow with "dPort = 80" takes significantly more resource than other flows. If NetFuse identifies this flow aggregation, any HTTP request traffic (including the upcoming ones) will be controlled, and the other applications in the network will be protected. Note that this list of rules includes both spatial ($P1$ to $P6$) and temporal ($P7$ to $P9$) flow properties, and we can easily obtain the required information in OpenFlow (more on this in Section IV). However, it by no means covers all possible aggregation conditions and can be customized by the operators based on the domain knowledge.

**Algorithm 1**: `maxAgg(`$\pi$`)`

```
1  π* = {P|P ∈ π, S(P) > th};
2  while |π*| > 1 do
3      select P from π*, foreach P' ∈ π* do
4          if not (P ≤ P' or P' ≤ P) then
5              P'' = PP';
6              if P'' ∉ π* and S(P'') > S(P) and
               S(P'') > S(P') then
7                  π* ← π* ∪ P'';
8      if max{S(P)|P ∈ π*} > S(P) then
9          π* ← π* − {P};
10 if π* == ∅ then
11     P* ← flows;
12 else
13     P* ← the only aggregation in π*
```

**Threshold-based aggregation** Some reasonable aggregation rules may be based on a predefined threshold, *e.g.*, $P7$, $P8$, and $P9$. Using $P8$ from Table I as an example, if we want to partition flows by duration $d_f$, we need to set the threshold value $d_{th}$, and aggregate the flows with shorter duration ($d_f < d_{th}$) into the "short-lived" aggregation and the other flows into the "long-lived". We find an appropriate threshold by choosing the value that separates the aggregated flows by the largest distances. The distance definition can be determined given the historical measurement statistics during normal (and heavy-load) operations. For example, if the flow duration follows a power law distribution, we define the distance between threshold $d_{th}$ and aggregated flows $F$ as: $\min_{f \in F} (\log(d_{th}) - \log(d_f))$. Then we sort the flows in $F$ by $\log(d_f)$, find $i^* = \arg\max_i (\log f_{i+1} - \log f_i)$, and set $\exp((\log f_{i^*+1} + \log f_{i^*})/2)$ as the threshold. For different flow properties, we use different techniques (*e.g.*, clustering algorithms) to determine the threshold for the aggregation condition.

### *B. NetFuse Approach: Multi-Dimensional Aggregation*

In Table I, each aggregation is based on a single flow property. In practice, the overloading can be caused by specific applications at specific network regions, corresponding to a combination of multiple aggregation conditions on different flow properties. For example, if the overloading is caused by concurrent attacks against a database, then the corresponding aggregation may be "start time $< 30$ seconds" and "dPort $=$ $1433$". If we aggregate the flows by either start time or dPort, we cannot identify the right set of flows. Thus, a key problem is how to find the correct rule combination.

We develop a breadth-first search algorithm with branch pruning to enumerate the multi-dimensional flow aggregations (Algorithm 1). The input of the algorithm is $\pi = \{P_1, P_2, P_3, ...\}$, which includes the list of reasonable aggregations to be tried out. As an example, suppose $P_1 =$

$\{\{f_1, f_2\}, \{f_3\}\}$ is a flow aggregation based on start time, in which $f_1, f_2$ are "old" flows and $f_3$ is a "new" flow; $P_2 = \{\{f_1\}, \{f_2, f_3\}\}$ is another flow aggregation based on dPort, in which $f_1$ is a flow to port 22 and $f_2, f_3$ are flows to port 80. The algorithm generates a new aggregation $P_3$ by combining $P_1$ and $P_2$ together: $P_3 = \{\{f_1\}, \{f_2\}, \{f_3\}\}$, in which $f_1$ is an old flow to port 22, $f_2$ is an old flow to port 80, and $f_3$ is a new flow to port 80. If such a new aggregation $P_3$ has higher score $S(P_3)$ than $P_1$ and $P_2$, then the algorithm considers $P_3$ better and keeps it in $\pi$. The algorithm continues to combine partitions to find the best flow aggregation. Note the "$\leq$" operation represents the *refinement* relation between two set partitions. The algorithm does not combine the two aggregations if one is a refinement of the other.

The computational complexity of Algorithm 1 is $O(NK^m)$, where $N$ is the number of active flows, $K$ is the number of pre-determined aggregation rules, and $m$ is the dimension of the optimal aggregation. In practice, the algorithm can finish in under 1s, which is sufficiently responsive for online detection and reaction. NetFuse applies this algorithm to find the best reasonable flow aggregation and the corresponding most likely overloading reason. If no aggregation shows overloading, the output of the algorithm may be just the flows without aggregation (this can also be viewed as the finest aggregation). Otherwise, the output of the algorithm may be either single- or multi-dimensional flow aggregation rule(s). NetFuse will classify all flows falling into the aggregation as overloading.

## III. Adaptive Control

### A. Basic Control

Once NetFuse finds the suspicious flows, it must control them to mitigate the overloading problems. For each identified overloading flow, NetFuse instructs the associated switches to reroute the flows to the NetFuse box. Rerouting, different from mirroring, frees the network resource originally occupied by the overloading flows. Then, NetFuse can delay or selectively drop packets of the redirected flows to reduce their rates. In Section IV, we discuss how to realize this functionality in an OpenFlow network.

### B. Toxin-Antitoxin Mechanism

The identified suspicious flows should not be treated equally. Due to potential false alarms, not all suspicious flows may be misbehaving. Even if abnormal flows are correctly found, different flows may have different impacts on the aggregate overloading behavior. In addition, traffic interdependencies may exist within the identified flows or between them and the other normal flows. Therefore, NetFuse should regulate the identified flows differently and adapt the reaction according to the system feedback.

We introduce an adaptive control mechanism, inspired by toxin-antitoxin biological systems. During the cell division, reproducible poison and a limited amount of antidote are both released; the ill-behaved child cells kill themselves by aggressive reproduction (together with the poison), while the well-behaved child cells control themselves and survive with the sufficient antidote. A similar mechanism was used by Mahajan *et al.* [13] to control high bandwidth aggregate flows in the Internet.

NetFuse applies the toxin-antitoxin mechanism in its control actions as follows. When it delays an overloading flow $f$, NetFuse tests the aggressiveness of $f$'s response. If $f$ reduces its rate, NetFuse also reduces the aggressiveness of the delay until it no longer delays $f$. Otherwise, NetFuse starts delaying $f$ more aggressively until it eventually fills the buffer, dropping all packets of $f$. In particular, for NetFuse control, if the target rate of $f$ is $r$ (*e.g.*, the average rate of other normal flows), and $f$'s current rate demand (measured at NetFuse box buffer queue) is $r_f$, the extra delay NetFuse puts on $f$ is: $(r_f - r) \times r_f \times \text{RTT}/r$. Assuming the flows employ TCP and thus the flow rate is inversely proportional to RTT, since $r_f \propto RTT$, the extra delay on $f$ intends to reduce flow rate from $r_f$ to $r$. In practice, well-behaving flows respond to the extra delay by lowering the rate to or under $r$. Misbehaving flows may continue at the higher rate, and NetFuse will continue delaying and potentially dropping their packets.

## IV. NetFuse Design

We implement NetFuse as a proxy device between the OpenFlow switches and the controller. Next, we describe how NetFuse monitors and controls network flows.

**Monitoring** NetFuse employs both *passive listening* and *active query* to collect necessary network information. As an intermediate relay, NetFuse intercepts all the control messages between the switches and controller, thereby obtaining a global view of the network.

Each switch sends a PacketIn message to the controller every time it receives a data packet that does not match any of its flow table entries. The controller replies with an FlowMod message to install a forwarding rule for the packet and the subsequent packets in the same flow. Both PacketIn and FlowMod messages encode key flow information: identifier (*e.g.*, source IP and port, destination IP and port), routing configuration (*e.g.*, VLAN tag, ingress interface, egress interface), and the flow start time. When the flow entry expires (triggered by a configurable timer encoded in the FlowMod), the switch sends to the controller a FlowRemoved message, which includes the expired flow's duration and volume. All these control messages are embedded in the normal operation of OpenFlow networks and thus can be efficiently utilized by NetFuse with little additional overhead.

In addition to passively listening to the control messages, NetFuse can also use the OpenFlow ReadState messages to periodically query the switches for network resource utilization and fine-grained flow information. The network utilization information includes the interface load and queue size. NetFuse also actively queries fine-grained information about the long-lived elephant flows, in particular the dynamic flow rates between the flow start time and the expire time. Inevitably, these active queries introduce overhead to the data path. To mitigate this problem, NetFuse sets the query frequency to be in proportion to the current network load. In addition, NetFuse

conducts more active queries only for suspected flows and in suspected network regions.

**Detection and action** Given the set of flows derived from control traffic monitoring, NetFuse applies the flow aggregation algorithm to identify the flows with overloading behavior. During normal operation, all the flow reports from the switches are relayed to the controller. In overloading situations, especially when the control network is heavily loaded, NetFuse filters and prioritizes the flow reports to offload to the controller. The control commands from the controller also go through NetFuse before getting installed in switches. When it detects network problems and identifies the responsible flows, NetFuse performs adaptive control on the identified flows by modifying or issuing new flow control rules to the switches. In cases where the deep packet inspection is required or when extra delays need to be injected into certain flows (as described in Section III-A), the identified flows are redirected to the NetFuse boxes for further processing.

The NetFuse boxes work as transparent devices between the controller and switches. From the perspective of switches, these devices act as the controller and issue actual commands to the switches to perform certain tasks. From the perspective of the controller, they are the network. Additionally, the NetFuse proxies also relieve the controller from heavy tasks such as flow redirection, delay injection, and packet blocking, thereby improve the scalability of the entire system.

## V. Preliminary Experiments

We built a prototype for NetFuse and deployed it our OpenFlow-enabled data center testbed. The testbed consists of 10 switches in a FatTree topology. We generate 500 random flows over 100 time slots and apply the flow aggregation algorithm at each time slot. We configured a few reasonable aggregation rules (as in Table I) for bootstrap, and we applied the simple algorithm in Section II to determine the threshold for each corresponding rule. When a link is overutilized, NetFuse identifies the overloading flows based on the best flow aggregation rule. We create multiple test cases with specific overloading flows and examine whether NetFuse can correctly detect the overloading behavior.

**Expt 1: single compromised flow** We start with the simplest case, in which we pick up one normal flow, and suddenly increase its rate at some time slot. Figure 2(a) shows that the suspicious flow uses more resources than the other flows and can be easily identified, even without aggregation.

**Expt 2: DDoS attack** We simulate a DDoS attack by injecting flows from different switches to an edge switch. Figure 2(b) shows the identified overloading aggregated flow obtained from combing the rules "frequency $< 4$" and "start time $< 30$ seconds". This aggregated flow includes all the DDoS attack flows but also mis-identifies normal flows that start at the same time as the attack. The rate of false alarm is $9\%$. The adaptive NetFuse reaction will eventually block the real bad attacking traffic and let the normal flows pass.

**Expt 3: routing misconfiguration** We introduce a routing misconfiguration and let around 50 new flows routed from a core switch $A$ to an aggregation switch $B$ and eventually to an edge switch $C$. Although none of the new flows has very high rate, the total load of all the rerouted traffic can overload some network links. Figure 2(c) shows the sampled network utilization, in which the core-to-aggregation link is overloaded.

Given the rate of each flow (Figure 2(d)), NetFuse needs to identify the new flows that are causing the overload. Let's consider first the flows that go through the overloaded core-to-aggregation link (Figure 2(e)) without any aggregation. It is very difficult to identify which flows are having overloading behavior. Once we perform aggregation on these flows by combining frequency and routing as aggregation rules, the detection becomes much easier. Figure 2(f)) shows the rate increase for flows aggregated by combining the criteria "frequency $< 4$" and "routing from switch $A$ to switch $B$ to switch $C$," which correspond to the rerouted new traffic.

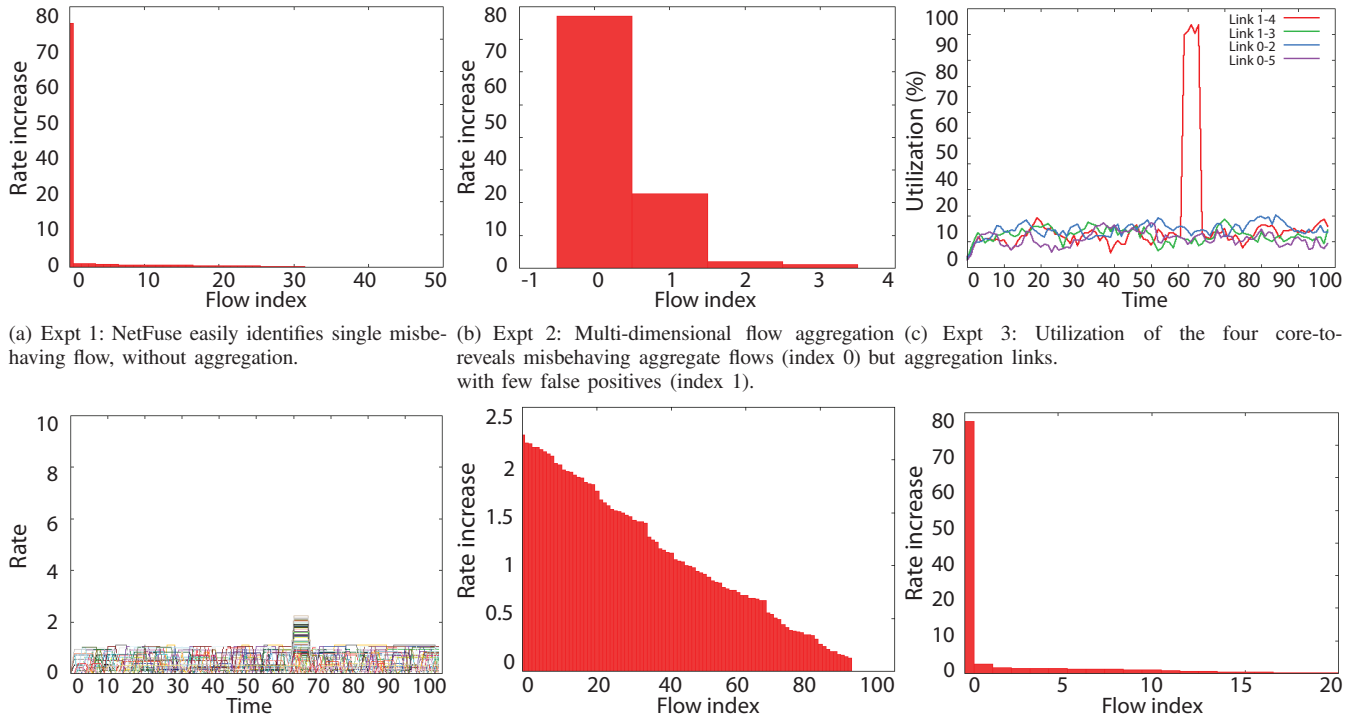
## VI. Discussion and related work

**Monitoring** NetFuse shares a similar philosophy with the recent work of Jose *et al.* [14] and of Al-Fares *et al.* [15]. Both works actively read switch counters unlike NetFuse which relies largely on passive measurements and uses active querying only when control traffic is insufficient. In addition, Jose *et al.* focus exclusively on detection, unlike NetFuse who attempts to also limit the effects of surges.

**Flow aggregation** We use multi-dimensional aggregation to find the problematic flows. This is similar to the hierarchical heavy hitter (HHH) problems [16], [17] in traditional IP networks. However, the proposed algorithms may not be directly applied to data center networks since the natural IP address hierarchy may not always exist, especially in such topologies with decoupled physical and logical addressing schemes such as VL2 [1] and Portland [18]. Even if the spatial properties of the flow can be hierarchically aggregated, the temporal properties of the flow still do not fit in the HHH algorithms.

**Scalability** Two mechanisms help scale NetFuse: network statistics inference based on the OpenFlow control messages and controller offloading using the NetFuse middlebox. The presence of a large number of flows triggers many control packets and can overwhelm NetFuse. Previous measurements show that having a large number of flows is feasible for medium-sized OpenFlow networks with a powerful controller or a collection of controllers. For example, controllers in networks of 100 switches, with new flows arriving every $10\mu$s, may have to process up to 10 million PacketIn messages per second [19]. Even if the controller were distributed, NetFuse could still capture control traffic and synchronize network information using a mechanism similar to FlowVisor [20].

**Deployment** In practice, fully reactive OpenFlow networks are rare. A more realistic deployment setting for NetFuse is a hybrid network where not all switches trigger control traffic (because they are not OpenFlow enabled or because they have rules pre-installed). In such environments, the current NetFuse monitoring mechanisms need to be coupled with SNMP-based techniques and flow sampling mechanisms. Moreover, in such hybrid environments, NetFuse will be deployed at a




(a) Expt 1: NetFuse easily identifies single misbehaving flow, without aggregation.

(b) Expt 2: Multi-dimensional flow aggregation reveals misbehaving aggregate flows (index 0) but with few false positives (index 1).

(c) Expt 3: Utilization of the four core-to-aggregation links.

(d) Expt 3: Rates for all flows.

(e) Expt 3: NetFuse cannot easily identify the misbehaving flows when there is no flow aggregation.

(f) Expt 3: With flow aggregation, NetFuse identifies the aggregation responsible for overload.

Fig. 2. Experimental results.

few vantage points instead of ubiquitously in the network. In such cases, NetFuse and other traditional network overload protection approaches can be complementary to each other.

## VII. Conclusion

We presented NetFuse to protect against network overloading problems in OpenFlow data center networks. NetFuse uses (1) passively-collected OpenFlow control traffic to identify active network flows, (2) multi-dimensional flow aggregation to find the flow clusters that are suspicious, and (3) a toxin-antitoxin mechanism to adaptively limit the rate of suspicious flows without severely affecting the false positives. We demonstrated NetFuse in our lab OpenFlow testbed and showed that it can detect DDoS attacks and routing misconfigurations.

Generated by IEEEtran.bst, version: 1.12 (2007/01/11)

## References


[1] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a Scalable and Flexible Data Center Network," in *ACM Sigcomm*, 2009.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *ACM Sigcomm*, 2010.

[3] "Why amazon's cloud titanic went down," http://money.cnn.com/2011/04/22/technology/amazon_ec2_cloud_outage/index.htm.

[4] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Anderson, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained tcp retransmissions for datacenter communication," in *ACM Sigcomm*, 2009.

[5] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: A data center network virtualization architecture with bandwidth guarantees," in *ACM CoNEXT*, Nov. 2010.

[6] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM Sigcomm*, Aug. 2011.

[7] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM Sigcomm*, 2011.

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *USENIX NSDI*, 2007.

[9] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *ACM Sigcomm*, 2005.

[10] K. Yamanishi and Y. Maruyama, "Dynamic syslog mining for network failure monitoring," in *ACM KDD*, 2005.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM Sigcomm CCR*, vol. 38, pp. 69–74, 2008.

[12] "IBM and NEC team up," http://www-03.ibm.com/press/us/en/pressrelease/36566.wss.

[13] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," *SIGCOMM CCR*, vol. 32, no. 3, pp. 62–73, 2002.

[14] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches," in *USENIX Hot-ICE*, 2011.

[15] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *USENIX NSDI*, 2010.

[16] G. Cormode, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in data streams," in *In Proc. of VLDB*, 2003, pp. 464–475.

[17] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications," in *ACM IMC*, Oct. 2004.

[18] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *ACM Sigcomm*, 2009.

[19] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM IMC*, 2010.

[20] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the test-bed," in *USENIX OSDI*, 2010.